View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

THE NETWORK IS THE DATABASE

GUO YUZHI

NATIONAL UNIVERSITY OF SINGAPORE

2004

THE NETWORK IS THE DATABASE

BY

GUO YUZHI
(B.E., RUC, PRC)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

I am very grateful to my supervisor, Dr. Stéphane Bressan, for his clear guidance and faith in letting me pursue these new directions. Also he has contributed many valuable suggestions. Each time when I got stunted in my research he would give me new ideas and new directions. From him I learned how to do research and how to think when encountering problems. I cannot thank him enough for all what he has done for me, and feel very lucky that I have been able to work under his instructions.

The second person to whom I should express my sincere gratitude is Ms.Tan Kim Luan. She gave me invaluable advice and guidance on revises of my thesis.

I should thank my friends, Chen Chao and Shen Qinghua. My research also benefits from the discussions and cooperation with them. Without their help, it is impossible for me to achieve the research results so quick.

Last I'd like to express my thanks to my families and my fiancee Wang Ying, for the love and encouragement they have been giving me all along these years of my life.

Many thanks go to the National University of Singapore for a Research Scholarship and School of Computing for providing us with a pleasant working environment and first class resources.

Contents

Acknowledgements **i**

Contents **ii**

Summary **vi**

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 2

1.3 Organization 4

2 Background and Related Works **5**

2.1 Communication by Broadcast in Databases 5

2.1.1 Pull v.s. Push 6

2.1.2 Broadcast Disks 7

2.2 Concurrency Control in Databases 8

2.2.1 Concurrency Control in Traditional Databases 9

2.2.2 Concurrency Control in Distributed Databases 11

2.3 Concurrency Control in Mobile Databases and Broadcast Disks . . . 13

2.3.1 Concurrency Control in Mobile Databases 13

2.3.2 Concurrency Control in Broadcast Disks 14

2.4 Consistency of Cache in Client 16

3 Concurrency Control for Simple Update with Locking in Broadcast Disks 21

3.1 Overview 21

3.2 Client-server Model 22

3.3 Broadcast Channel with Updates and Locking 23

3.4 Broadcast Disks with Updates and Locking 27

3.5 Broadcast Disks with Client Cache, Updates, and Locking 32

3.6 Summary 35

4 Performance Analysis for Simple Update with Locking in Broadcast Disks 37

4.1 Experimental Set-up 37

4.2 Comparative Performance Analysis 38

4.2.1 Broadcasting versus Client-server 38

4.2.2 Broadcast Disks versus Broadcast Channel 42

4.2.3 Client Cache and Replacement Policies 45

4.3 Summary 46

5 Concurrency Control for Database Transaction with Locking in Broadcast Disks **48**

5.1 Overview 48

5.2 Concurrency Control in Broadcast Disks Environment without Cache in Client 49

5.2.1 Correctness Criteria in Broadcast Disks 49

5.2.2 *APPROX* – Update with Broadcast Disks in [SNSR99] . . . 52

5.2.3 Basic Locking Update with Broadcast Disks 55

5.2.4 Discussion 59

5.3 Update with Cache in Client 60

5.3.1 Server 61

5.3.2 Client 61

5.4 Summary 64

6 Performance Analysis for Database Transaction with Locking in Broadcast Disks **65**

6.1 Experimental Set-up 65

6.2 Comparative Performance Analysis 66

6.2.1 Updates in Clients with Different Concurrency Control Mechanism 66

6.2.2 Updates in Clients with Broadcast Disks Environment 68

6.2.3 Updates in Clients with Broadcast Disks Environment with Cache in Client 70

6.3 Summary 74

7 Conclusions and Future work **75**

7.1 Contributions 75

7.2 Future work 76

Bibliography **78**

Summary

Broadcasting fits the requirement of more and more modern applications, such as traffic report systems, weather forecast and stock activity monitoring systems, for the distribution of data. This is due to the nature of data, to the ubiquity and multiplicity of devices consuming and producing data converging with the development of a global network infrastructure.

Is broadcasting a viable mechanism for the complete management of persistent objects? Can a broadcasting system accepting updates maintain a reasonable performance while guaranteeing concurrency control? To address these questions and possibly answer some of them, our research focuses on the design and analysis of a peer-to-peer persistent object management system leveraging the network infrastructure. We consider an architecture in which the data are not stored on the disks of the computers participating in the system - or if so, in caches, buffers, and logs, but cyclically broadcast on the network.

In this thesis we study the design and evaluate the performance of update model controlled by a basic locking mechanism for broadcast disks architecture that involves replication on the broadcast channel and in the clients' caches.

As a first step, we present a simple update and locking mechanism for a broadcast disks model with cache. The new concurrency control mechanism combines the advantages of both the locking in conventional database and the broadcast disks environment. There can now be, on the broadcast channel, several copies of the same object available for reading as well as one copy available for update by one identified

client or/and exclusively one already updated copy not available for reading or writing. Clients can read the previous version of an object while the current version is being updated. This proposal constitutes a sound basis for the implementation of concurrency control strategies for the interleaved execution of transactions. The models we devise and study can support the implementation of schedulers or lock-based concurrency control strategies.

Furthermore, we extend our research from simple operations to transaction processing in the broadcast disks. We combine locking mechanism and multi-version concurrency control to guarantee the correctness of the concurrency among the transactions. Our proposed models based on broadcasting are efficient and yield better performance than the basic locking mechanism when the opportunities for sharing objects for read are high: high ratio of read over write operations, large number of clients, and skewed distribution of popularity of objects. Thus, broadcasting remains a viable alternative even in the presence of updates by the client. The better performance of our models is particularly important since broadcasting is a candidate architecture for many new applications involving new devices and networks.

In summary, our contributions are the proposal of a new concurrency control mechanism combined by the basic locking and the broadcast disks in the peer-to-peer environment. From the theoretical analysis and results of experiments, we have presented the validation and efficiency of our concurrency control mechanism, thus paving the way for the future work in this environment.

List of Figures

3.1 Broadcast Channel Architecture 23

3.2 Client's Monitoring Algorithm in the Broadcast Channel Model 26

3.3 Server Loop for a Broadcast Channel Model of Model 3.2 26

3.4 Client's Monitoring Algorithm in the Broadcast Disks Model 28

3.5 Server Loop for a Broadcast Disks Model 29

3.6 Processing of Read and Write in Model 3.3 31

3.7 Client's Monitoring Algorithm in the Broadcast Disks Model with Cache 33

3.8 Server Loop for a Broadcast Disks Model with Cache 35

4.1 Throughput at $\theta = 0.5$ and 64 clients, with varying ρ 39

4.2 Throughput at $\theta = 0.5$ and 64 clients, with varying $1/\rho$ 40

4.3 Throughput at $\theta = 0.5$ and ρ=1, with varying number of client 41

4.4 Throughput at ρ=4 and 64 clients, with varying θ 41

4.5 Throughput at $\rho = 4$ and 64 clients, with varying number of object 42

4.6 Throughput at θ_r=1, θ_w=0.5 and 64 clients, with varying ρ 43

4.7 Throughput at θ_r=0.5, θ_w=1 and 64 clients, with varying ρ 44

4.8 Throughput at θ_r=1, θ_w=0.5 and 64 clients, with varying $1/\rho$ 44

4.9 Throughput at 64 clients, θ_r=θ_w=0.5 and Cache size of 5, with varying ρ 45

4.10 Throughput at 64 clients ,θ_r=θ_w=0.5 and *LRU*, with varying ρ 46

5.1 Venn Diagram for Classes of Schedules 51

5.2 Server Loop for Our Model 57

5.3 Client's Algorithm in Our Model without Cache 58

5.4 Client's Algorithm in Our Model with Cache 62

6.1 Response time at θ=1 and client=64, with varying ρ 67

6.2 Response time at θ=0.5 and client=64, with varying ρ 67

6.3 Response time at θ=1, ρ=4 and client=64, with varying number of data object . 68

6.4 Response time at θ=1 and client=64, with varying ρ 69

6.5 Response time at θ=0.5 and client=64, with varying ρ 69

6.6 *LRU*: Response time at client=64, with varying ρ for different θ 70

6.7 *LIX*: Response time at client=64, with varying ρ for different θ 70

6.8 *LRU*: Response time at client=64, with varying θ for different ρ 71

6.9 *LIX*: Response time at client=64, with varying θ for different ρ 72

6.10 Response time at client=64 and ρ=16, with varying θ 72

6.11 Response time at client=64 and ρ=1, with varying θ 73

List of Tables

2.1 Comparison of related work in broadcast disks 17

3.1 Description of the data structure of the algorithm 25

4.1 Parameter of the experiments on simple update model 38

5.1 Comparison of *APPROX* and our model 60

6.1 Parameter of the experiments on transaction process 66

Chapter 1

Introduction

Broadcasting emerges as a significant social and technical phenomenon and fits the requirement of more and more modern applications, such as traffic report systems, weather forecast and stock activity monitoring systems, for the distribution of data. This is due to the nature of data (multimedia data or more generally streamed data, e.g. movies and radio programs [VI95]), to the ubiquity and multiplicity of devices consuming and producing data (personal computers in peer-to-peer environment, mobile phones and portable devices, and sensors, e.g. [ZGE01]) converging with the development of a global network infrastructure. Broadcasting has combined some distributed database systems as a whole data resource to serve large populations of clients.

1.1 Motivation

Is broadcasting a viable mechanism for the complete management of persistent objects? For read-only requests from clients, it can outperform the conditional client-server system. However, can a broadcasting system accepting updates maintain a reasonable performance while guaranteeing concurrency control? Can recovery be

implemented when the data is mainly on the volatile broadcast channel? To address these questions and possibly answer some of them, our research focuses on the design and analysis of a concurrency control mechanism, by which the database will allow updating from the clients, in a peer-to-peer persistent object management system leveraging the network infrastructure. We consider an architecture in which the data are not stored on the disks of the computers participating in the system - or if so, in caches, buffers, and logs, but cyclically broadcast on the network.

Broadcasting architectures in general and broadcast disks in particular outperform traditional client/server architectures when many clients read data from few servers. Yet several issues arise when the broadcasting model allows updates by the clients. These issues, most of which are related to the control of the concurrent access (isolation and consistency), are rooted in the asynchronous nature of the broadcast model and in the replication of data on the broadcast channel and in the caches of the clients.

Although several concurrency control mechanisms have been proposed and studied for broadcasting, to our knowledge, no one has studied the performance of a basic locking mechanism. Specifically, we study concurrency control and recovery in an environment of autonomous peers communicating with an Internet protocol with replication on the network and in the peers' caches.

1.2 Contribution

In this thesis we study the design and evaluate the performance of update model controlled by a basic locking mechanism for broadcast disks architecture that involves

replication on the broadcast channel and in the clients' caches.

As a first step, we present an update and locking mechanism for a broadcast disks model with cache. The new concurrency control mechanism of this model combines the advantages of both the locking in conventional database and the broadcast disks environment. There can now be, on the broadcast channel, several copies of the same object available for reading as well as one copy available for update by one identified client or/and exclusively one already updated copy, not available for reading or writing. Clients can read the previous version of an object while the current version is being updated. This proposal constitutes a sound basis for the implementation of concurrency control strategies for the interleaved execution of transactions. The models we devise and study can support the implementation of schedulers or lock-based concurrency control strategies.

Furthermore, we extend our research from simple data operation into transaction processing in the broadcast disks environment. We combine locking mechanism and multi-version concurrency control to guarantee the correctness of the concurrency among the transactions. Our proposed models based on broadcasting are efficient. They incrementally yield better performance by the basic locking mechanism when the opportunities for sharing objects for read are high: high ratio of read over write operations, large number of clients, and skewed distribution of popularity of objects. Thus, broadcasting remains a viable alternative even in the presence of updates by the client. The better performance of our models is particularly important since broadcasting is candidate architecture for many new applications involving new devices and networks.

Finally, we have conducted the comparative experiments and can observe the features of the broadcast disks environment and the advantages and disadvantages of the concurrency control in this environment, thus paving the way for the future work in this environment.

In summary, our contributions are the proposal of a new concurrency control mechanism combined by the basic locking and the broadcast disks in the peer-to-peer environment. We will present the validation and efficiency of our concurrency control mechanism by the theoretical analysis and results of experiments shortly.

1.3 Organization

The remaining chapters of the thesis are organized as follows:

In Chapter 2 we discuss the background and related work in the fields of broadcast disks and concurrency control in database.

In Chapter 3 we present the models of broadcast read and point-to-point update.

In Chapter 4 we analyze the performance of our proposed model and compare with the traditional client-server model.

In Chapter 5 we describe the model of concurrency control for transaction in broadcast disks environment.

In Chapter 6 we analyze the performance of our proposed model and compare with the APPROX model in [SNSR99] .

In Chapter 7 we conclude this thesis and outline future work.

Chapter 2

Background and Related Works

Our research focus is on the concurrency control of database system in the broadcast disks environment. In this chapter we discuss the background of communication and the broadcast disks environment in database. After the review of the concurrency control in traditional database, we investigate the related work on concurrency control in mobile database and broadcast disks. Finally, we discuss the problems of the consistency of the cache in clients.

2.1 Communication by Broadcast in Databases

The network technique has led to an increase in many new database applications. However, with the increasing number of the accessible networked data sources, some asymmetric problems [AFZ97], such as the network asymmetry and the imbalance between the number of client and the number of server, will interrupt these new applications in the distributed data source. The main reason is that the communication method in the traditional database system does not match the environment of the distributed data source and may not yield good results.

Broadcast technology makes it possible to disseminate the data to a large pop-

ulation of clients in many forms of application, such as traffic information system, software distribution and entertainment delivery. Thus data broadcast acts as a primary role in dissemination-based applications. Here, data distributed in a small number of sources can be broadcast to a larger number of clients with similar interest.

2.1.1 Pull v.s. Push

A pull-based and request-response style of operation is used in the traditional client-server database system. Thus, a client has to send a request message to the server to access a data object. When the server has received the request, it will manage the whole database and response to the client. Although the client plays an active role in the system, it still has its drawbacks when the system has more clients and fewer servers. This in turn will generate a huge volume at the server and thereby creates a bottleneck in the system.

In order to reduce the bottleneck, push-based data dissemination method has been proposed. In contrast to the pull-based data delivery method, the push-based mechanism pushes the data needed by the client, into a broadcast channel and the client monitoring the channel had to wait for the incoming data without any request. For example, [AFZ97] combines the pull-based and push-based system and analyzes the performance and impact by the ratio of pull to push processes. However, this system only allows for the read request, it does not process the conflicts between the different operations, such as read and update, on the same data object.

2.1.2 Broadcast Disks

[AAFZ95] constructs an architecture for database management by the multi-level broadcast disks. The architecture is based on the communication network. It includes the broadcast server and several clients. Broadcast disks is a technique of the push-based system for replying the request of the clients of the system.

The responsibility of the broadcast server is to store the whole database and to broadcast the data to the clients through the broadcast channel. Because the data are pushed on the broadcast channel, the channel can be viewed as some disks. According to the differences in the allocated sizes of each disk and different access probabilities of data, the broadcasting data are then assigned to the different sizes and speed disks. Hence, there are some relatively different frequencies among the different disks and the data on the faster disks are broadcast more often than the data on slower disks. The improved version of broadcast is the multi-disks broadcast in which there is no variance in the inter-arrival time for each broadcast page for each disk. Based on the information about the preference of the broadcasting data, the broadcast server computes the relative frequency and creates the multi-disks broadcasting schedule by replicated the hotter pages.

Clients in this system have the capacity of communicating to the server by the broadcast channel, and they also have their local memory. When they want to obtain some data to meet their needs, they will firstly retrieve the data from their local cache. If the needed data is not found, the client will then be monitoring the broadcast channel and wait for the desired data to arrive. As the broadcast server

broadcasts the preferred data according to the client, the individual client should store the data in a place where the local frequency of access is greater than the frequency of broadcast in their local memory. Therefore there should be an optimal data replacement policy for the client to manage the cache. When the client wants to update the data that is being broadcast on the channel, they will complete the process locally, send the updated request to the server and wait for the commit or abort message.

2.2 Concurrency Control in Databases

Concurrency control is one of the essential techniques for a database system. More and more concurrency control mechanisms are being studied so as to improve the performance of the different database systems. In any database, a transaction is the basic unit of the whole system and consists of some operations which are reading or writing database objects. In order to improve the performance of the system, transactions are allowed to execute interleaved. A set of transactions arranged by certain order is called a schedule [RR00] and a schedule in which transactions are executed serially is called a serial schedule. A schedule in which transactions produce a result which is the same as a serial schedule is called a serializable schedule. Two operations on the same data conflict if at least one of the operations is a write. These conflicts can destroy the consistency of the database if without the appropriate concurrency control mechanism. Thus in different databases', developing the better mechanism of concurrency control is the main concern.

2.2.1 Concurrency Control in Traditional Databases

The aim of the management of any system is to improve the performance or throughput of the system. For database system, the performance can be improved by allowing transactions interleaved executing [RR00]. For instance, when a long transaction is waiting for a certain page, another short transaction, which can be completed quickly, will be allowed to execute. Although transactions are allowed to executed interleaved, the state of database must maintain a consistent state, that is, the result of executing a serial or serializable schedule on a consistent database is in a consistent state. Thereby we must guarantee the consistent state of the database when executing interleaved transactions.

Lock-Based Concurrency Control

For a centralized database system, "a lock is a mechanism used to control access to database objects ([RR00], p.17)". Every transaction must follow the locking rules of the locking protocol to ensure the serializability and reliability of database. As usual, there are two kinds of lock: shared lock and exclusive lock. Depending on the kind of the operation used to read or write, transaction must obtain the relevant lock. Share lock can be held by different transactions on the same data object but exclusive lock is only obtained by one transaction on one data object.

Non-Locking Concurrency Control

Besides locking, there are alternative approaches to concurrency control. According to light contention of the conflict in database system, the overhead of requesting locks and following the lock policy are a waste of time.

For the database system in which most transactions will not conflict with others, a permissive method called *Optimistic Concurrency Control*, the transaction first reads or writes the object in a local private workspace. When the transaction wants to commit, it must be checked by the server whether there will be a conflict with any other concurrently executing transactions. If there is no conflict, the private workspace are then copied to the database, otherwise, the workspace will be cleared and transaction will restart. This method, however, is not suitable for database requiring high-performance of transaction processing [Tho98].

Timestamp is often used as the order of transaction execution for lock-based and optimal concurrency control, so as to ensure the serializable schedule and check for possible conflicts among transactions. Timestamp can also be used by another concurrency control policy [RR00], which can eliminate the deadlock problem for concurrency control. Transaction is assigned a unique timestamp and each data is assigned a write-timestamp and a read timestamp. The conflicting operations will then be resolved by timestamp order. Whenever a transaction restarts, its timestamp should be assigned a new timestamp.

In multiversion concurrency control, the system maintains some versions (or copies) of data items in order to improve the performance of database system. Each write operation on a data item creates a new version of this item. Each read operation will be told by the system which version of item it should read. The first advantage of multiversion concurrency control is to avoid rejecting the operation that arrive too late, which means transaction can read the most recent version of the data item. The second advantage is its lower cost, as a database system needs

a recovery algorithm which also needs these versions of the data items. However, the high cost of maintaining the multiversions is the storage space. The versions have to be flushed periodically and this process creates a problem in multiversion concurrency control that needs to be addressed. Since the multiverisons of the data item cannot been seen by user transactions, a multiversion history is correct if and only if it is equivalent to a serial one-version history.

2.2.2 Concurrency Control in Distributed Databases

In a standard distributed database system, concurrency control is a mechanism for synchronizing concurrent transactions in such a way that the consistency of the database is maintained while maximum degree of concurrency is achieved at the same time. There are three basic algorithms based on the locking system, in the distributed database system: centralized (primary site) 2PL, primary copy 2PL and distributed 2PL[MT99].

For centralized 2PL there is only one 2PL lock manager(scheduler) in the distributed system and all lock requests of the transactions from the client are issued to the central scheduler. However, this algorithm is similar to the centralized database system and caused more communication on the data distributed on other servers. Hence, the concurrency control must solve the distributed replication of data conflicts. The solution is to place a scheduler at each site to manage the concurrency control. The primary copy 2PL means assigning a copy of data as a primary and it will handle all the requests on the set of data. The distributed 2PL means the site of any copy could act as the scheduler of the data for any transaction.

There are mainly three schemes for discovering the conflicts in distributed sites, *Read-lock-one, write-lock-all (ROWA)*, *majority locking*, *primary copy locking. Read-lock-one, write-lock-all (ROWA)* means that a transaction may obtain a read-lock on any copy of data and if a transaction wants to write any data, it should be obtain write-lock on all copies of the data. In the *majority locking* strategy, a transaction must obtain read or write locks on a majority of the copies of the data read or written by the transaction. The third scheme is the *primary copy locking*, which means all locks for a data are requested at the site of the primary copy assigned by the system.

For a distributed database system, [JMR98] proposed an efficient protocol of multiversion concurrency control. It stores at most three versions, it also guarantees that the update transactions cannot interfere the read transaction, the read transactions need not request a lock and add the information into data. And the advancement of the version is asynchronously managed by the system. For each site in the distributed system, there are only two versions of the data items initially. One is for read and the other is for update. Periodically, a version advancement process runs to make the new version and to refresh the read copy. For any read operation, it can read the read copy without any lock request and the write transaction should follow the 2PL lock protocol. During the version advancement process, it starts at the updated version and may create a third version to update, and thus ensuring that there is no active transaction on the read version, which then delete it and make the old update version to read version, hence the version advancement is completed. This multiversion concurrency control protocol is suitable for the distributed sys-

tem, in which queries does not need to be the very latest data but consistent data is highly desirable.

2.3 Concurrency Control in Mobile Databases and Broadcast Disks

2.3.1 Concurrency Control in Mobile Databases

Advancement of wireless communication technology pushes the progress in the applications of database system and it is widely used into the distributed database system. Although the basic property has not been changed and the concurrency control policy of the distributed database can still be utilized in mobile distributed database, the performance is decreased by the characteristics of the wireless network. Based on the characteristics or limitation of the wireless network, such as limited bandwidth and frequent disconnection, the aims of improving the mobile system led to a change in how to reduce the overhead of the communication for concurrency control and the number of the restarting transaction due to blocking.

In the mobile distributed database system, transactions of client can be shipped to the relevant server in order to reduce the overhead of the communication. For the mobile network, concurrency control cannot use the policy utilized in distributed database, as the overhead of resolving the conflict is very expensive. [LKTL00] proposed a similarity-based distributed two phase lock protocol. Similarity means two operations of two concurrent transactions, which are either both reads or both writes and the object values are still similar. By this similarity of two transactions, they are allowed to interleaved execute without effect on the integrity and consistence of the database. The similarity is derived from the semantic analysis of the transaction

and is utilized to resolve the concurrent conflict, which means that the conflicting operations of two transactions occur similarly and are allowed to execute concurrently.

According to the limitation of the mobile network, the traditional "pull-based" data delivery approach has been weakened. In this asymmetric communication environment, a new data delivery approach based on "push" is proposed. The data will then be pushed to the client by the server's advantage in bandwidth and the client will retrieve the item from the broadcast channel according to their needs.

[LCC99] considered the broadcast process as a transaction, checked the concurrent conflict between broadcast and update transaction instead of detecting conflict between update and read transactions from client, which is called *Update First with ordering* (UFO). UFO protocol consists of execution phase and update phase. During the execution phase, the conflict between transactions is resolved by a conventional concurrency control and the write operation updates the new value into a private workspace. When all transactions have been completed, the data broadcast process will be checked for data conflict. By comparison of write set and read set of broadcast process, the system will then re-broadcast the inconsistent data to the client.

2.3.2 Concurrency Control in Broadcast Disks

Based on the previous research in [AAFZ95], which focused on the improvement of the broadcast schedule in the read-only broadcast disks environment, [AFZ96a] proposed the broadcast disks model with both read and update on the database.

Although the update is only processed on the server, the experiment results can improve on the consistency between the cache and server and act as the preliminary model of ours in the thesis. Two methods of ensuring the consistence of cache, invalidation and propagation, are proposed and compared. Their experiment results show that the broadcast disks model is robust in the presence of updates.

In the late nineteen eighties, a group of researchers proposed a radically new architecture for database management systems intended to increase the throughput. In this architecture, called the Datacycle [TFBW92, GHW87], the server cyclically broadcasts database objects on a broadcast channel. The clients monitor the broadcast channel for the objects they want to manipulate. Other communications with the server such as the management of update and concurrency control utilize an upstream channel from the clients to the server. The Datacycle model use serializability as the correctness criterion to maintain the consistency of the database, but it is very expensive to communicate with the server for all operation requests in the broadcast disks environment.

In [SNSR99] a control matrix is used to resolve the concurrent conflicts. The matrix size is determined by the database size, that is, for a database of n data items, a matrix of size $n \times n$ is used. For each broadcast cycle, the control matrix is broadcast together with the data item. Before the client reads the data from the broadcast channel, it will check the control matrix to perform a consistency check. This method requires the read operation to retrieve the consistent data and write the data locally. At the end of the transaction, the whole transaction, including all of the read and write operations and the cycle identifier, will be sent to the server

for committing.

Another concurrency control protocol, which is called STUBcast(Sever Timestamp and Update Broadcast Supported Concurrency), is designed for broadcast-based transaction in [YH01b]. Two new correctness criteria— Single Serializability and Local Serializability — are supported by STUBcast. Single Serializability ensures that all update transactions and any single read-only transaction are serializable. Local Serializability requires having all update transactions in the whole system and all read-only transactions at one client site are serializable. The two new correctness criteria are both weaker than global serializability and easier to achieve. STUBcast allows both read-only transaction and update transaction. The broadcast server divides broadcast operations into primary broadcast, which broadcasts data items to client using certain broadcast algorithm, and update broadcast, which is inserted into the primary broadcast with committed data item update. There are three components in the STUBcast protocol: client side read-only serialization protocol(RSP), client side update tracking and verification protocol(UTVP), and server side server verification protocol(SVP). They only abort the read-only transaction, which cannot be serialized with the committed update transaction and ensure Single Serializability and Local Serializability for all committed transactions in the system.

Table 2.1 shows the comparison between the related work and our models.

2.4 Consistency of Cache in Client

When the cache technical is used at client, there must be a protocol between the client and the server to ensure that the cache at client remains consistent with the

Model	[AFZ96a]	Model in Chapter 3	[TFBW92]	[SNSR99]	[YH01b]	Model in Chapter 5
Style of process	O	O	T	T	T	T
Style of request in client	R	R/W	R/W	R/W	R/W	R/W
Style of broad-cast disks	Multi-disk	Multi-disk	Flatdisk	Flatdisk	Flatdisk	Multi-disk
Conflict	No	Yes	Yes	Yes	Yes	Yes
Method on solv-ing conflicts	–	Lock	Optimistic Concur-rency Control	Optimistic Concur-rency Control	Time-stamp	Lock
Cache	Yes	Yes	No	Yes	No	Yes
Method of Cache consis-tency	Invadation report	Invadation report	No	Time con-straint	No	Matrix

Table 2.1: Comparison of related work in broadcast disks

database at server. [FCL97] presented a taxonomy providing a unified treatment of proposed caches consistency algorithms for client-server database system. Based on whether detecting or avoiding access to stale cache data, the taxonomy proposed the numerous dimensions of the design space for transactional cache consistency algorithm. Six algorithms presented by previous papers are described and analyzed according to the tradeoffs inherent in the design choices of the taxonomy.

[WN90] presented two algorithms to address the cache consistency problem. One is a modification of two-phase locking and consists of adding new lock modes–cache locks. The other is based on notify lock. In the first algorithm, there are three status

of the cache lock for data in client cache: *Cache lock, Pending update lock and Out-of-date lock*. Upon updating on the data in cache, the client has to request lock from server before accessing it. After receiving a response of committing request, the client marks the data in updating list with *out-of-data* lock. When the server receives a request of committing from client, the server will update the different status of cache lock to share or exclusive lock according to different situations. A message indicating whether the transaction has committed or aborted and a list consisting of the data updated will be sent back to client. Based on the notify lock, the server will send notification of the updates to all clients periodically. Until the server receives the commit request, it may still be sending notification to the client that could abort the transaction. A handshake, which is implemented by assigning a sequence number to every message sent from the server, is required between server and client to ensure that the client has accessed the most recent notification message.

In a mobile computing environment, it is a useful technique for reducing the contention on the narrow bandwidth of the wireless channel to cache frequently accessed data. However, expensive communication cost required by transactional cache consistency strategies in traditional client-server system is not appropriate in a mobile computing environment. [SL99] proposed a protocol, called OCC-UTS (*Optimistic Concurrency Control with Update TimeStamp*), to maintain transactional cache consistency in mobile environment. Each client keeps two lists for a current transaction: readset, including data items read by this transaction, and updateset, including data items updated and new value by this transaction. A client performs a read operation either by the cache or by requesting from server. After the client

sends the readset and updateset to server for committing, it will continuously listen to the channel for CommitList and AbortList and check whether it is committed or aborted. The server constructs *Invalidation Report* to achieve serializability of mobile transaction by optimistic concurrency control in [KR81] and broadcasts it to maintain the cache consistency of data item in clients.

For Invalidation Report, [Cao02] proposed an improvement to address the ineffective utilization problems. Firstly, a small fraction of the essential information called updated invalidation report(UIR) is replicated several times and broadcast to clients within an IR interval. Then the client can refresh its cache without waiting for the next IR and reduce the latency of query. [Cao02] also utilized the prefetch technique to improve the performance of the caching at client and the cache hit ratio. Clients prefetch the data item that are most likely to be used in the future by counter for each data item. The counter is increased by one when the data is requested by a client, and the counter is decreased by one if a client replaces its replication by some other data items. Based on the counter, the server can find which data is hot and broadcast their update to client. If a counter is zero, the data item need not to be included in the Invalidation Report to save the bandwidth.

[YH01a] proposed a new cache strategy for real-time transaction to meet their time deadline in broadcast environment. Given the number, the sequence and the ID of the data requests for a transaction, a cache policy called Largest First Access Deadline Replaced(LDF) is used as a solution for the caching and pre-fetch strategy. The access deadline of a data on any transaction is defined as an estimated latest time for this data to be accessed. Any cache maintains an Access Deadline Table

including data ID, Access Deadline and First Access Deadline for all data items on all current transactions it involves in. The First Access Deadline is the minimum among all access deadlines of this data in all current transactions. When a data is accessed from the broadcast channel, the value of this data in the Access Deadline Table should be modified and subsequently, the client will check whether the data cached with the largest value of First Access Deadline has a larger value than that of the current data. If this is so, such a data will be replaced by the current data out of the cache. This policy keeps the data that is needed more urgently for current transactions to meet their deadline.

Chapter 3

Concurrency Control for Simple Update with Locking in Broadcast Disks

In this chapter we present the reference client-server model and three incremental variants of our model. Without loss of generality we consider in the discussions below a single server and several clients. Ultimately every computer in the network can play both roles simultaneously thus yielding a peer-to-peer architecture.

3.1 Overview

In our proposed model, we attempt to combine the advantages of both conventional database and broadcast disks environment as the new model. In order to maintain consistent state of the database, update requests must be controlled by the database server, and read requests can be processed of the broadcast mechanism. Thereby, the performance of the model should be improved.

3.2 Client-server Model

For reference here and in our performance analysis, we first recall the traditional client-server model. We later refer to this model as the *client-server* model.

The server manages objects. Such objects are often pages containing data elements. If a client needs to read or update a data element, it reads or updates the object or page that contains it.

Client In the client-server model clients must request to the server the objects for a read or a write operation. Read- and write-requests are sent by clients to the server by an upstream channel.

Server The server controls the concurrent accesses by maintaining a lock table. Each entry in the lock table indicates, for any given currently accessed page, the type (read or write) and number of locks on the object as well as a queue of requests waiting on this object. If no potential conflict is detected in the lock table, the server delivers the requested object to the client on a downstream channel, otherwise the request is queued. If the server receives a read-request for an object, a read-lock is added to the lock table for this object if the object is free or locked by a read-lock. If the object is locked by a write-lock the request is queued. If the server receives a write-request for an object, a write-lock is added to the lock table for this object if and only if there is no previous lock. Locks are released when the server receives release-read-lock or release-write-lock notices indicating that the client has completed the operation. The server then attempts to process request in the queue.

Release-write-lock notices may consist of the modified object itself.

Discussion Such a standard mechanism is the basis for the implementation of concurrency control strategies for the interleaved execution of transactions. For instance it is the basis of the implementation of the classic two phase locking strategies guarantying serializability and recoverability properties for transactions.

If one is not concerned with transaction support then a simpler locking mechanism can be used that simply prevents the creation of diverging versions of the same object. This simpler mechanism always grants read requests. It records write locks and grants write-requests only when there is no write lock. The server, however, must guarantee that reads and writes are atomic operations at its side. In the sequel, we consider a standard locking mechanism with read-write and write-write conflicts.

3.3 Broadcast Channel with Updates and Locking

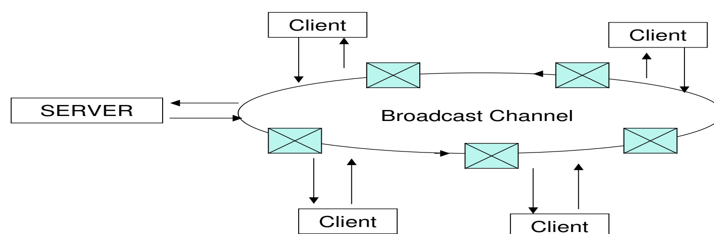

Figure 3.1: Broadcast Channel Architecture

We now replace the upstream channel used by the clients to send their requests and the updated objects by a single cyclic channel, which we call the broadcast

channel. Figure 3.1 illustrates the architecture of a cyclic broadcast channel with clients and server placing and monitoring objects and messages on the channel. The model we propose is now similar to the one of a broadcast disk [AAFZ95] except that we still consider that clients send both read- and write-requests. We later refer to this model as the *broadcast channel* model.

For the sake of simplicity of the presentation we assume that the broadcast channel is a ring that allows the cyclic circulation of messages (requests, notices, and objects). Although we do further discuss the issue in this chapter, an arrangement of server and clients in a ring is not strictly necessary provided the realization of weaker assumptions about messages eventually reaching clients and server. In particular, we need not assume that the order of messages sent on the network is maintained since synchronization is done at the server's side when the messages return. This is important as it allows clients to hold notices possibly contribute the implementation of a scheduling strategy. The reader may notice that other proposal for broadcasting architecture with updates and concurrency control often make stronger assumptions about the order of messages on the channels.

Client The client algorithm is presented on Figure 3.2 and the description of variables of all the algorithms is shown in Table 3.1. A client needing to read an object sends a read-request to the server on the broadcast channel. It then monitors the broadcast channel until the object to be read is at hand. The client reads the object from the channel.

Name	Description
my	the active object generating in the client and including the read or write request
page	the active object being broadcast in the channel and accessed by the client
my.kind	the kind of request, reading or writing, in the active client
my.data_id	the identity of the data requested by the client
my.client_id	the identity of the active client
my.rset	the set of data identity read by the active client
my.wset	the set of data identity written by the active client
page.destination	the recipient identity of this page
page.tag	the kind of page, such as response page, broadcast page, request page
page.data_id	the data identity in this page

Table 3.1: Description of the data structure of the algorithm

A client needing to update an object sends a write-request to the server on the broadcast channel. The client monitors the broadcast channel until the object to be updated is at hand. The client removes the object from the channel, updates it, and places it back on the channel. A client is allowed to update an object if it is tagged with a note identifying the client as the recipient. A client is not allowed to update an object tagged with a note identifying a different client as the recipient.

Server The server algorithm is presented on Figure 3.3. The server maintains a lock table identical to the one used in the client-server model. When the server receives a read- or write-request it grants the request according to the lock table, i.e. updates the lock table and sends the object on the broadcast channel, else it queues the request. In the case of a write-request the object sent is tagged with the identifier of the recipient (i.e. the client requesting the page for update). Read- and

Algorithm

```
1.  While(1 > 0) { /*monitoring loop*/
        /*get the page from the broadcast channel*/
2.      page = (TRANSACTION *) get((HEAD *) broadcastchannel);
3.      If(my.kind='w' && page.destination == my.client_id && page.tag == 'w'){
        /*update request*/
4.          wSend(page); /*Send response message to the server*/
5.          Commitcouter++;
6.          break;
7.      }
8.      Else /*Broadcast reading*/
9.          If(my.data_id == page.data_id && page.tag == ' ' && my.kind='r'){
10.             rSend(page);/*place page back to the channel*/
11.             Commitcouter++;
12.             break;
13.         }
14.     rSend(page);/*place page back to the channel*/
15. }
```

Figure 3.2: Client's Monitoring Algorithm in the Broadcast Channel Model

Algorithm

```
1.  page = (TRANSACTION *) get((HEAD *) broadcastchannel);
2.  switch(page.tag){
3.      case ' ': /*read page*/
4.          Unlock(page); /*release read lock*/
5.          Break;
6.      case 'u': /*update response message from the client*/
7.          Unlock(page); /*release write lock*/
8.          Break;
9.      case 'q': /*request from client*/
10.         Lock(page) /*concurrency control*/
11.         Break;
12. }
```

Figure 3.3: Server Loop for a Broadcast Channel Model of Model 3.2

write-locks are released as untagged and tagged pages, respectively, come back to the server. The pages themselves serve as lock-release notices.

Discussion There can be several copies of an untagged object, i.e. an object requested for read, on the broadcast channel. It is not necessary to indicate the recipient of the copy of an object requested for read and any client having requested an object for read can read any copy of the object. This is a potential source of higher throughput. There can however only be one copy of a tagged object, i.e. an object requested for update, exclusively of untagged objects on the broadcast channel. This is a potential bottleneck that we try to avoid with the extension presented in the next subsection.

3.4 Broadcast Disks with Updates and Locking

Except for a possible simultaneous copy of untagged objects, our model so far does not allow the replication of objects on the broadcast channel. In other words, our previous broadcast disks model had a single disk spinning at the lowest possible speed and we now present a model extending a multi-disk and multi-speed broadcast disks model following [AAFZ95] to update and locking. We later refer to this model as the *broadcast disks* model.

Client The client algorithm is presented on Figure 3.4. A client needing to read an object need not send a read-request to the server on the broadcast channel. It just monitors the broadcast channel until the object to be read is at hand.

As previously, a client needing to update an object sends a write-request to

Algorithm

```
1.  While(1 > 0) { /*monitoring loop*/
        /*get the page from the broadcast channel*/
2.      page = (TRANSACTION *) get((HEAD *) broadcastchannel);
3.      If(my.kind='w' && page.destination == my.client_id && && page.tag == 'w'){
        /*update request*/
4.          wSend(page); /*Send response message to the server*/
5.          Commitcouter++;
6.          WaitforAcknowledge(my.data_id);
7.          break;
8.      }
9.      Else /*Broadcast reading*/
10.         If(my.data_id == page.data_id && page.tag == ' ' && my.kind='r'){
11.             rSend(page);/*place page back to the channel*/
12.             Commitcouter++;
13.             break;
14.         }
15.     rSend(page);/*place page back to the channel*/
16. }
```

Figure 3.4: Client's Monitoring Algorithm in the Broadcast Disks Model

the server on the broadcast channel. The client monitors the broadcast channel until the object to be updated is at hand. The client removes the object from the channel, updates it, and places it back on the channel. The client must wait for an acknowledgement before it can read the same object again. This prevents it from reading an older version still on the broadcast channel. Otherwise, a client is allowed to read or update an object if it is tagged with a note identifying the client as the recipient. A client is not allowed to read or update an object tagged with a note identifying a different client as the recipient. A client is allowed to read any untagged object.

Server The server algorithm is presented on Figure 3.5. The server sends one or more untagged copies of each object on the broadcast channel according to a pre-determined policy (see [AAFZ95] for details) thus simulating several disks spinning

Algorithm

```
1.  page = (TRANSACTION *) get((HEAD *) broadcastchannel);
2.  switch(page.tag){
3.        case ' ': /*Broadcast*/
4.              If(IsOldVersion(page)!=True)/*collect old version pages*/
5.                    rSend(page);
6.              Break;
7.        case 'q': /*update request from the client*/
8.              If(Lock(page)==Ture){ /*concurrency control*/
9.                    Response(page.client_id); /*Response to client*/
10.                   SetOldVersion(page.data_id); /*Set old version page*/
11.             }
12.             Break;
13.       case 'u': /*response page from the client*/
14.             Unlock(page); /*release write lock*/
15.             While(CollectAll(page.data_id)!=True)
16.                   SetWait(page.data_id); /*wait for all old pages*/
17.             SendAck(page.data_id);/*broadcast acknowledge*/
18.             While(True&&Ack!=True)
19.                   SetWait(page.data_id); /*wait for acknowledge*/
20.             Broadcast(page); /*broadcast new version page*/
21.             Break;
22. }
```

Figure 3.5: Server Loop for a Broadcast Disks Model

a different speed (the more copies the higher the speed).

In the previous model of broadcast disks in [AAFZ95], the server allocates the data items into different disks according to the different access probabilities of each data item. However, it is only for the read-only request of the client. In our broadcast disks model, we found that the data for more frequently read should be replicated more copies and the data for more frequently update should be only one copy in the broadcast channel, which means the more frequently read data should be allocated in the faster disks and the more frequently updated data should be allocated in the slower disks. Thereby, we order all data items by the ratio of read-probability to write-probability and allocate them into the disks with different speed. The greater ratio the data has, the faster speed disk it is allocated.

The server maintains a lock table in which it records, for each object being used, the write-locks and the number of untagged copies currently on the broadcast channel. Write-locks are set as write-requests arrive. The copy counter is incremented as an untagged copy is sent on the channel and decremented as an untagged copy comes back to the server. When the server receives a write-request it grants if there is no write lock for the requested object and then sends the object on the broadcast channel, else it queues the request. The object sent is tagged with the identifier of the recipient (i.e. the client requesting the page for update). Write-locks are released as tagged pages come back to the server. The server resends every untagged object on the broadcast channel unless there is a write lock for the object. If there is a write lock, untagged copies cannot be resent. An untagged copy can only be sent after the counter for the object has reached zero, i.e. all untagged copies have come back to the server, the updated page has come back to the server, and an acknowledgement of update has been sent to the client performing the update and has returned. When the condition is met the server can start resending the untagged copies of the new version of the object.

Discussion Figure 3.6 illustrates a sequence of messages in the case of an update. The client sends on the broadcast channel a write-request for an object. The server, after receiving the request, locks the object and sends a tagged copy (the tag indicating the recipient client) on the broadcast channel. The copy reaches its recipient which performs the update and places the (tagged) updated object on the channel. The server waits for the updated objects and all the untagged copies to

return. It then sends an acknowledgement to the client. When the acknowledgment comes back the server can resume the broadcasting of the object, now in its updated version.

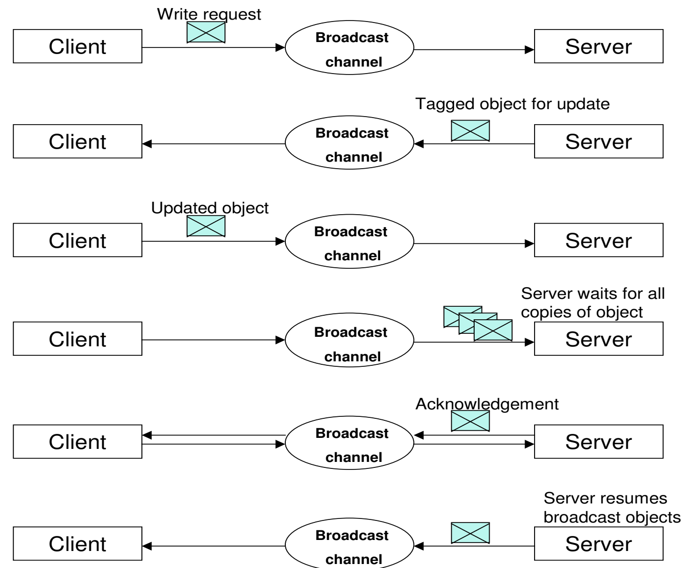

Figure 3.6: Processing of Read and Write in Model 3.3

This policy guaranties that there is on the broadcast channel only one version of an object that can be read. As before the pages themselves serve as lock-release notices.

The reader notices that the model has no read-lock. Although the untagged copy counter plays a similar role to the one of the read-locks, the model does not seem to be able to cater for read-write conflicts. Indeed the counter is not involved in the granting of write-request. We remember that no synchronization among the clients is assumed. In particular there is no global clock and time. Furthermore the clients do not send read-requests. We can therefore freely consider that every read-

request corresponding to a read made between the granting of a write request and the moment at which the updated page has arrived at the server and all the untagged pages have come back has virtually been sent before the write-request. We can also consider that the actual read occurs before the write. Since there are no read-locks, we can consider that prior to resuming the sending of the updated objects, the server releases all the read-locks before it releases the write-lock. Therefore the system is free of read-write conflicts.

There can now be, on the broadcast channel, several copies of the same object available for read as well as one copy available for update by one identified client or and exclusively one already updated copy not available for read or write. Clients can read the previous version of an object while the current version is being updated.

3.5 Broadcast Disks with Client Cache, Updates, and Locking

To complete the model we now consider the clients' caches. We now assume that clients may perform read and write operations directly on the data in their cache. We later refer to this model as the *broadcast disks with cache* model.

Client The client algorithm is presented on Figure 3.7. A client needing to read an object need not send a read-request to the server on the broadcast channel. If the object is in the cache it is used. Otherwise the client monitors the broadcast channel until the object to be read is at hand. The client removes the object from the channel, reads it, and places it back on the channel. When an object is read from the channel it is also loaded in the cache. An object currently in the cache may be

sacrificed according to the replacement policy. We later compare two policies: *LRU* (Least Recently Used) as used in [AAFZ95], and *LIX* which has been introduced in [AFZ96b]. *LIX* is an efficient constant time approximation of *PIX* [AAFZ95]. *PIX* maintains different queues for each disk in the multi-disk model and takes into account the broadcast frequency.

Algorithm

```
1.  If(my.kind=='w') /*send update request to server*/
2.      Request();
3.  If((my.kind=='r')&&(InCache(my.data_id)==True)) /*the needed data is in cache*/
4.      Commitcouter++;
5.  Else{
6.      While(1 > 0){ /*monitoring the broadcast channel for needed data */
7.          page = (TRANSACTION *) get((HEAD *) broadcastchannel);
8.          switch(page.tag){
9.              case ' ': /*page for reading*/
10.                 If((my.kind=='r')&&(my.data_id==page.data_id)){
11.                     Commitcouter++;
12.                     RefreshCache(page.data_id); /*refresh the cache*/
13.                     my.action=1;
14.                 }
15.                 Break;
16.             case 'w': /*response from the server*/
17.                 If(page.destination == my.client_id){
18.                     wSend(page); /*send the updated data to server*/
19.                     my.action=1;
20.                     Commitcouter++;
21.                     RefreshCache(my.data_id); /*refresh the cache*/
22.                     WaitAck(my.data_id); /*wait for acknowledge*/
23.                 }
24.                 Break;
25.             case 'i': /*invalidation message from server*/
26.                 If(InCache(page.data_id)==True)
27.                     OutCache(page.data_id); /*delete the stale data out of the cache*/
28.                 Break;
29.         }
30.         rSend(page);
31.         If(my.action==1)
32.             break;
33.     }
34. }
```

Figure 3.7: Client's Monitoring Algorithm in the Broadcast Disks Model with Cache

As previously, a client needing to update an object sends a write-request to the

server on the broadcast channel. The client monitors the broadcast channel until the object to be updated is at hand. The client removes the object from the channel, updates it, and places it back on the channel. The updated object is also loaded in the cache. An object currently in the cache may be sacrificed according to the replacement policy. The updated object is temporarily pinned in the cache. It is unpinned when the client receive acknowledgement of the update on the server and of the guarantee that no older copies of the object can be read. This acknowledgement comes in the form of an invalidation message for this object sent by the server.

A client is allowed to read or update an object if it is tagged with a note identifying the client as the recipient. A client is not allowed to read or update an object tagged with a note identifying a different client as the recipient. A client is allowed to read any object in the cache and any untagged object on the broadcast channel.

Server The server algorithm is presented on Figure 3.8. The server functions as in the previous subsection: upon receiving the updated object, it waits for all copies of the object to come back and sends an invalidation message, which also acts as a acknowledgement in previous model, to all clients and waits for the message to come back before it releases the write-lock and resumes the broadcasting of the tagged copies of the new version of the object.

Discussion Similarly to the model in the previous subsection this policy guaranties that there is on the broadcast channel and in the caches, except in the cache of the client having performed an update, at most one version of an object that can be read. The client performing the update must however wait until it receives

Algorithm

```
1.  page = (TRANSACTION *) get((HEAD *) broadcastchannel);
2.  switch(page.tag){
3.        case ' ': /*Broadcast*/
4.              If(IsOldVersion(page)!=True)/*collect old version pages*/
5.                    rSend(page);
6.              Break;
7.        case 'q': /*update request from client*/
8.              If(Lock(page)==Ture){ /*concurrency control*/
9.                    Response(page.client_id); /*Response to client*/
10.                   SetOldVersion(page.data_id); /*Set old version page*/
11.             }
12.             Break;
13.       case 'u': /*response message from the client*/
14.             Unlock(page); /*release write lock*/
15.             If(Collectall(page.data_id)==True&&isback==True)
16.                   Broadcast(page); /*broadcast new version page*/
17.             Else
18.                   SetWait(page.data_id); /*wait all old pages and invalidate message*/
19.             Invalidate(page.data_id);/*broadcast invalidate message*/
20.             Break;
21.       case 'i': /*invalidation message comes back*/
22.             InvalidateBack(page.data_id);/*collect invalidate message*/
23.             Break;
24. }
```

Figure 3.8: Server Loop for a Broadcast Disks Model with Cache

the invalidation report before it unpins the updated object and before it can let the replacement policy sacrifice the updated object from the cache. This prevents a subsequent read of an older version of the updated object.

3.6 Summary

In this chapter, we proposed a modification on the communication channel in the conventional concurrency control model. By the Point-to-Point method the server communicates their clients to notify them whether their request of lock is granted. We used the broadcast channel instead of the point-to-point communication in the server. The client can retrieve the needed objects from the broadcast channel under

the mechanism of concurrency control, in which the object can be read by all clients in the broadcast channel and the object for update only can be accessed by its requester. Furthermore, we investigated the different variants of the broadcast disks model with cache in client. In the next chapter we will evaluate the performance of these models and compare performance of two different caching policies in the broadcast disks by simulation experiments.

Chapter 4

Performance Analysis for Simple Update with Locking in Broadcast Disks

We empirically evaluate the performance of the models we proposed and compare it to the performance of the conventional client-server model in this chapter. We refer to our model as the hybrid model for it combines broadcast and point-to-point communication.

4.1 Experimental Set-up

For the purpose of the performance evaluation we simulate the models using the CSIM [JH99] discrete event simulator. The simulator measures performance in logical time units. We assume that it takes one unit of time to the server to process one incoming message on the up- or downstream channel and to reply by sending one object, request, or notice on the downstream channel (or the broadcast channel). The performance metrics we use is the throughput, i.e. the number of (read and write) operations executed per unit of time. The higher the throughput, the better the performance. We subsequently plot the throughput in number of operations per

5000 units of time.

Item Number	30, 50, 70, 90, 110
Client Number	8, 16, 32, 64
θ	1, 0.8, 0.6, 0.5, 0.4, 0.2, 0
ρ	1, 2, 4, 8, 16

Table 4.1: Parameter of the experiments on simple update model

To measure the relative performance of these models and illustrate their respective behavior we vary several parameters in our experiments. The parameter ρ indicates the ratio of read- over write-requests for any object. In our experiments it varies from one (there are as many read- as there are write-requests in average for any given object) to sixteen (there are sixteen times more read- than there are write-requests for any object). The parameter θ controls the probability for any object to be requested (either read- or write-request following the previous parameter ρ). θ is the parameter of a Zipfian distribution of the form: $p_i = (1/i)^\theta$. A value of zero for θ indicates a uniform distribution while a value of one indicates a skewed distribution (few objects are very frequently accessed). The server serves many distinct objects. Finally we vary the number of clients from 8 to 64. The successive values used in the experiments are reported in Table 4.1.

4.2 Comparative Performance Analysis

4.2.1 Broadcasting versus Client-server

To confirm our approach and validate our simulation with respect to the previous work on broadcast channels we first verify that the comparative performance of the broadcast channel model versus the client-server model is maintained in the

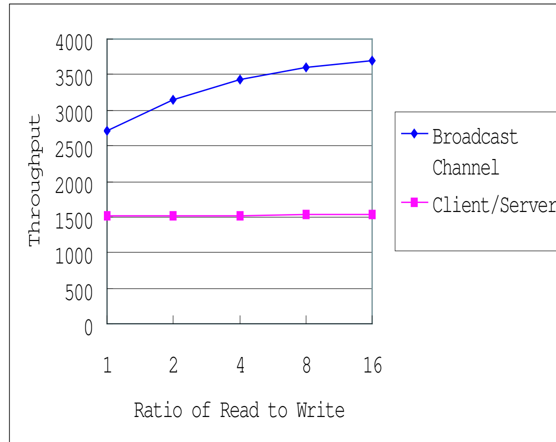

Figure 4.1: Throughput at $\theta = 0.5$ and 64 clients, with varying ρ

presence of update with locking. It is the case provided there are sufficiently enough opportunities for sharing objects on the broadcast channel for reading among the clients and few chances of operations being queued and delayed because of locks.

If the read-write ratio is sufficient, the broadcast channel model outperforms the client-server model even in the presence of updates and locks. As the ratio of read to write operations increases, the opportunity for sharing objects for read-operation increases for the broadcast channel model. For 64 clients and a mildly skewed popularity distribution with θ equals 0.5, Figure 4.1 shows that, as the number of reads over the number of writes varies from 8 to 16 the performance of the broadcast channel model increases significantly while the one of the client-server model remains practically unchanged. For the purpose of verification, we also measured the performance of the two models when there are no updates. As previous work on the broadcasting indicates, the broadcast channel model outperforms the client-server model with throughputs of 7137 and 2468, respectively. Indeed, in the client-server model, the server can process every incoming request in one unit of

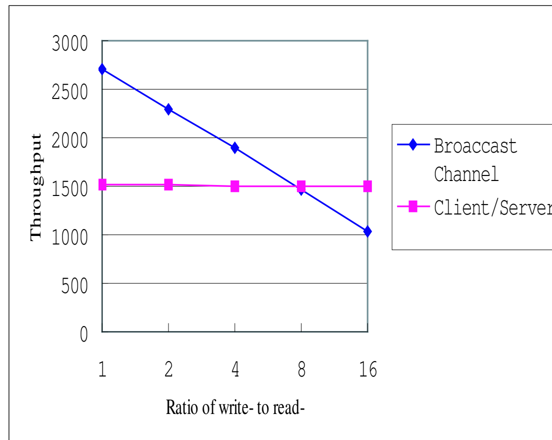

Figure 4.2: Throughput at $\theta = 0.5$ and 64 clients, with varying $1/\rho$

time. The server receives approximately 2500 requests and 2500 lock-release notices (one for each request) in 5000 units of time.

Moreover, we continue to increase the number of the writing operation and investigate the result of the models by varying $1/\rho$. Shown as Figure 4.2 we can find that the performance of client/server model is still changed litter, but the performance of the broadcast channel model has been decreased by the increasing of the ratio of write- to read-operation. When the ratio of write- to read- is greater than 8, the performance of client/server model is better than that of the broadcast channel model. The reason for that is the broadcasting for writing operation can not improve the performance of the broadcast channel model and only the read operation can be shared by the broadcast channel.

Similarly, as the number of clients increases, the opportunity for sharing objects for read-operation increases for the broadcast channel model. For a mildly skewed popularity distribution with θ equals 0.5 and a read-write ratio of 1, i.e. in conditions mildly favorable to the broadcast channel, Figure 4.3 shows that, as the number of

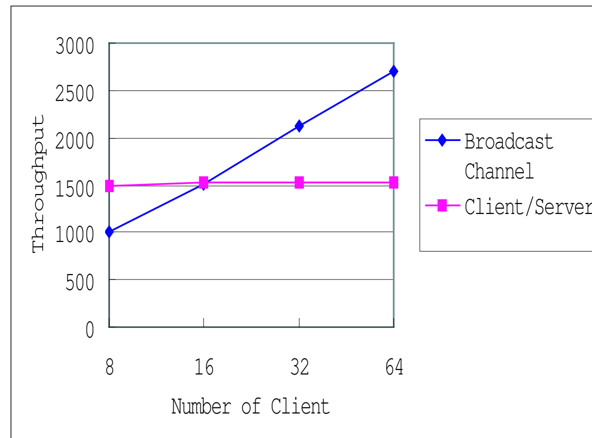

Figure 4.3: Throughput at $\theta = 0.5$ and ρ=1, with varying number of client

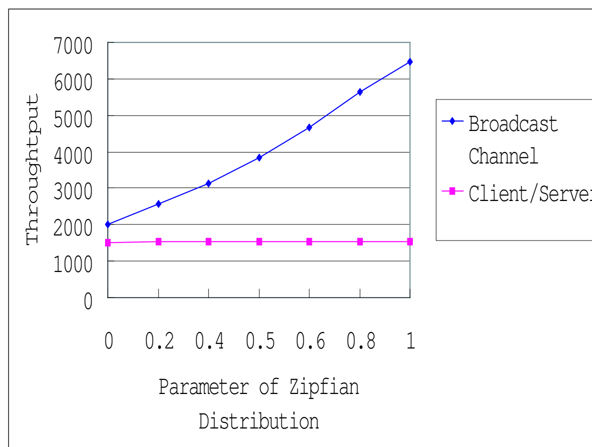

Figure 4.4: Throughput at ρ=4 and 64 clients, with varying θ

clients increases from 1 to 16, the performance of the broadcast channel model increases significantly while the one of the client-server model remains practically unchanged. The broadcast channel model outperforms the client-server model for 16 clients and more.

As the distribution of popularity of objects becomes skewer, the opportunity for sharing objects for read-operation increases for the broadcast channel model. However, there is potentially a higher risk of requests being queued and delayed

because of locks for both models. For 64 clients and 4 times more read operations than write operations, Figure 4.4 shows that, as the distribution of the popularity of objects becomes skewer, the performance of the broadcast channel model increases significantly while the one of the client-server model remains practically unchanged.

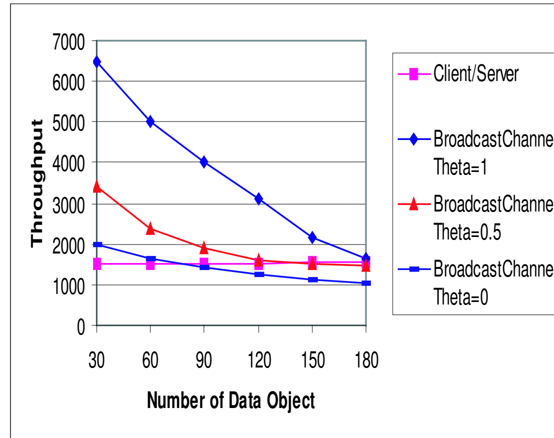

Figure 4.5: Throughput at $\rho = 4$ and 64 clients, with varying number of object

We also investigate the effect of the number of object on the server. From Figure 4.5 we find that the performance of the broadcast channel is decreased by the increasing of the number of objects. When the data object on the server become more and more, the data to be requested become dispersed. As a result, the opportunity of sharing becomes less and the performance of the broadcast channel model is decreased and the performance of client/server model is better.

4.2.2 Broadcast Disks versus Broadcast Channel

Broadcast disks improve on a read-only broadcast channel by simulating several disks spinning at different speeds. We compare in this subsubsection the broadcast channel model with several versions of the broadcast disks model with different combinations of speeds. We use three different sets of broadcasts, two of which

simulate three disks spinning at 5/3/1 and 7/4/1, respectively, and the third one a single disk.

In this group of experiments we use different distributions for the popularity of objects for read and write. This separation is important since, as the experiments demonstrate, the best performing model depends on these distributions.

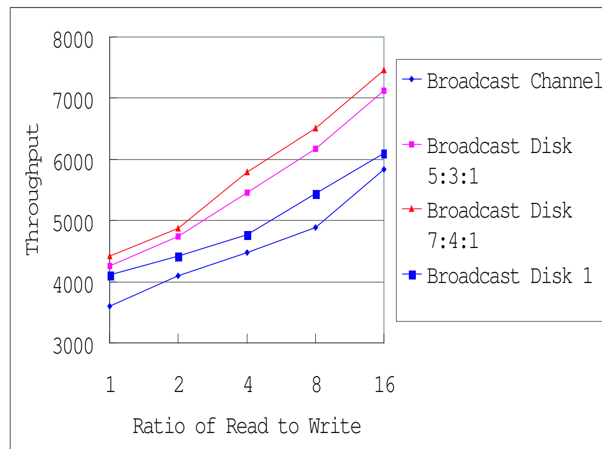

Figure 4.6: Throughput at θ_r=1, θ_w=0.5 and 64 clients, with varying ρ

Figure 4.6 shows the performance of the four models with a Zipfian popularity distribution for reads with parameter θ_r of 1.0 and a Zipfian popularity distribution for writes with parameter θ_w of 0.5. The broadcast disks models perform better than the broadcast channel model with disks speed of 7/4/1 yielding the best performance.

Figure 4.7 shows the performance of the four models with a Zipfian popularity distribution for reads with parameter θ_r of 0.5 and a Zipfian popularity distribution for writes with parameter θ_w of 1. Conversely to the prior experiment, under these distribution parameters, the broadcast disks models still perform better than the broadcast channel. However, conversely to the previous case, it is now the broadcast

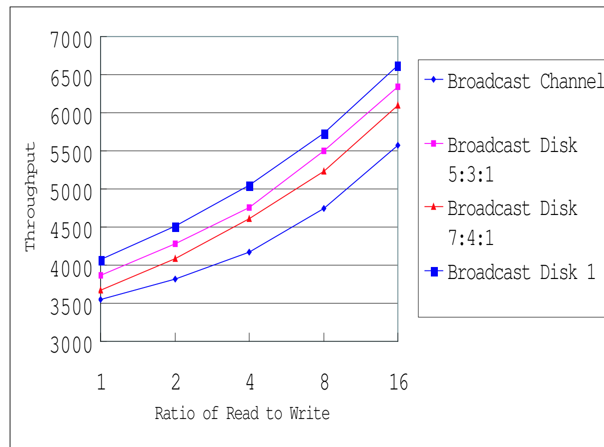

Figure 4.7: Throughput at θ_r=0.5, θ_w=1 and 64 clients, with varying ρ

channel model with disks speed of 7/4/1 that yields the worst performance.

Like previous experiments, we also study the effect of the ratio of write- to read-operation for our models. Shown as Figure 4.8 the performance of the flat disk configuration is better than other two models when the ratio is greater than 3. The reason for that is the more writing operations the more cost on synchronization of the data. As a result the flat disk model becomes the best configuration in this environment.

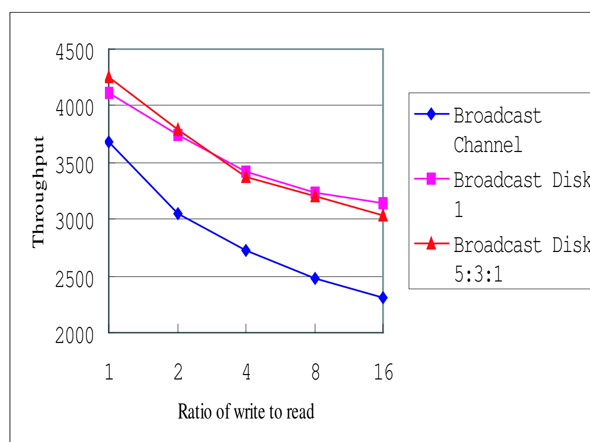

Figure 4.8: Throughput at θ_r=1, θ_w=0.5 and 64 clients, with varying $1/\rho$

The experiments confirm that broadcast disks require a good (a priori) knowledge of the distribution of the popularity of objects (we remark that it would be interesting to devise an adaptive strategy for the replication of objects). Our contribution, however, is a model for client update and server locking in the broadcast disks model in spite of the replication inherent to the concept of multi-speed broadcast disks. Under the above model, our experiments show that both popularity distributions for read and for write separately but not independently influence the performance.

4.2.3 Client Cache and Replacement Policies

The broadcast disks model with cache at the client side introduces an additional need for synchronization among the clients and the server: the cache inconsistencies invalidation. The experiment results are given for hot caches, 64 clients and $\theta = 0.5$.

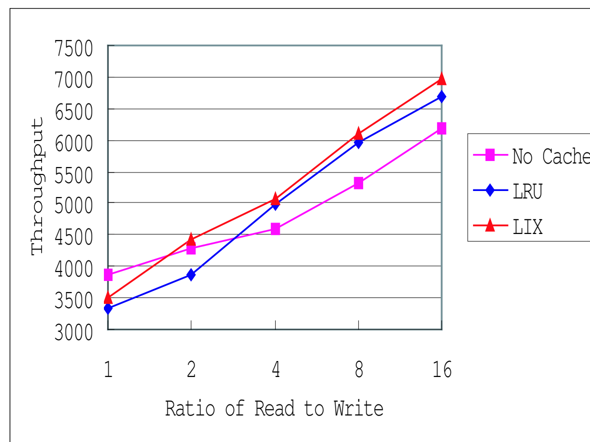

Figure 4.9: Throughput at 64 clients, θ_r=θ_w=0.5 and Cache size of 5, with varying ρ

We first show that the results of [AAFZ95] regarding the best performing replacement policy in a broadcast disks model with cache still holds in the case of a model with updates and locking. Figure 4.9 shows, for varying read-write ratio, the

performance of the broadcast disks model without cache and two broadcast disks models with *LRU* and *LIX* replacement policies, respectively. The figure shows that both models with cache outperform the model without cache. It also confirms that *LIX* performs slightly better than *LRU*.

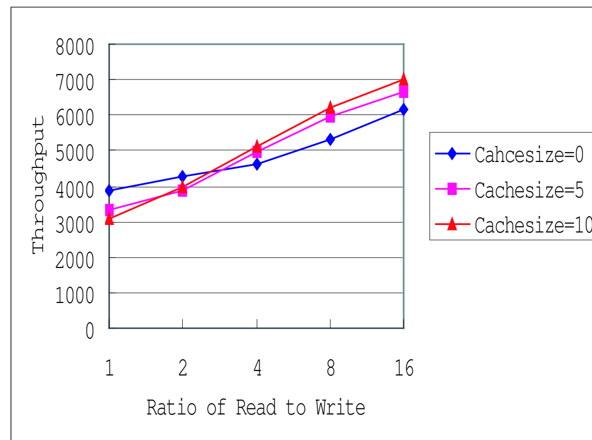

Figure 4.10: Throughput at 64 clients ,θ_r=θ_w=0.5 and *LRU*, with varying ρ

We next show that the cost of invalidation can be overcome by the gain of caching when there are sufficiently more read than write operations. Figure 4.10 shows, for varying read-write ratio, for 64 clients, and for popularity distributions of parameter θ of 0.5, that, when there at least three times more read than write operation, the best performance is obtained for the biggest client cache.

4.3 Summary

We have presented an update and locking mechanism for a broadcast disks model with client's cache. In order to introduce and study the features of our model we presented three different incremental versions: a model using a broadcast channel, a model using broadcast disks, and a model using broadcast disks and allowing caches

at the clients' side. This proposal constitutes a sound basis for the implementation of concurrency control strategies for the interleaved execution of transactions. The models we devised and studied can support the implementation of schedulers or lock-based concurrency control strategies.

The three models we have proposed are based on broadcasting. They incrementally yield better performance compared to the reference client-server architecture when the opportunities for sharing objects for read are high: high ratio of read over write operations, large number of clients, and skewed distribution of popularity of objects. We have indeed demonstrated that broadcasting remains a viable alternative even in the presence of updates by the client. This result is particularly important since broadcasting is candidate architecture for many new applications involving new devices and networks.

Chapter 5

Concurrency Control for Database Transaction with Locking in Broadcast Disks

In this chapter, we extend our research focus from the concurrency control of elementary update operation into transaction processing in the broadcast disks environment and we prove the correctness of this relaxed concurrency control mechanism in order to improve the performance of the model.

5.1 Overview

In our proposed model, we combine the control matrix and locking mechanism in the broadcast disks environment as the relaxed concurrency control model. In order to maintain consistent state of the database, update requests must be controlled by the locking mechanism in the database server, and the read requests can be checked by the control matrix for the whole transaction and met by the advantage of the broadcast mechanism. When the local memory is utilized in clients for improving the model performance, problems on data consistency arise. The data in the update set of the committed transaction must affect their replications in other client cache,

otherwise, data inconsistency will occur when another transaction access these data objects. In order to keep the clients' cache consistent with the update value on the server, the invalidated report is proposed to solve the inconsistency problem of replications in the clients' cache.

5.2 Concurrency Control in Broadcast Disks Environment without Cache in Client

Now, we propose a concurrency control model with broadcast disks. In this model, which is extended from the model in Chapter 3, one server and many clients are connected by a broadcast channel. All data objects are stored as a page in the server and will be broadcast to all clients by the server through the broadcast channel. Any client can generate a transaction consisting of the operations like read or update operation for any data object on the server and commit or abort any transaction.

In order to maintain the whole database consistency among these transactions of the clients, a control matrix like in [SNSR99] is proposed for the read-write conflict resolution and the locking mechanism is explored for the solution of the write-write conflict in broadcast disks. We will first discuss the correctness criteria of the models and then describe the functionalities of the model – *APPROX* in [SNSR99], and based on the *APPROX* model, our model will be represented.

5.2.1 Correctness Criteria in Broadcast Disks

Although the conflict serializable is the commonly accepted correct criteria for transactions and used in many database systems, generally, it is too expensive to maintain in broadcast disks environment. Higher communication costs with the server make

the conflict serializable inappropriate for the broadcast disks. Thereby, the alternative correctness criteria for transactions in broadcast disks should not only be weaker than conflict serializable, but also ensure the consistency of the database.

In *APPROX*[SNSR99] update consistency in [BC92] and optimistic concurrency control were proposed to maintain the consistency of the database. Update consistency in [BC92] is weaker than conflict serializable but is able to solve the read-write conflicts in the database system. However, optimistic concurrency control is also explored to maintain the serializability in their model. Optimistic concurrency control is a non-lock concurrency control, which basic premise is that most transactions will not conflict, and the timestamp acts as the criteria of the validation among the transactions. Thus there are some disadvantages of the model of [SNSR99]. First, while there are more update transactions in the model, the optimistic concurrency control will restart most of them and reduce the performance of the model. Second, the cost of identity of the global time in each client, which is used to check the validation of the transactions, is higher.

To solve these problems, a correctness criteria called One-copy Serializability in [PAB00] based on the multiversion concurrency control is proposed in our model. According to Figure 5.1 in [RR00] One-copy Serializability is one kind of view serializable and weaker than conflict serializable, but is able to ensure the correctness of the database and of the data object read concurrently by transactions in broadcast disks environment. Unlike the conflict serializable with the same conflict order, a schedule is One-copy Serializability if it is view equivalent to some serial schedule, that is, they have the same reads-from relationships and the same final writes. There

are one or two versions of each data stored in the server. If a data has two versions, one will be the version being broadcast, and the other will be the version updated by a transaction being committed. Once the transaction commits, this version becomes the unique committed version to be broadcast and the previous committed version will be deleted.

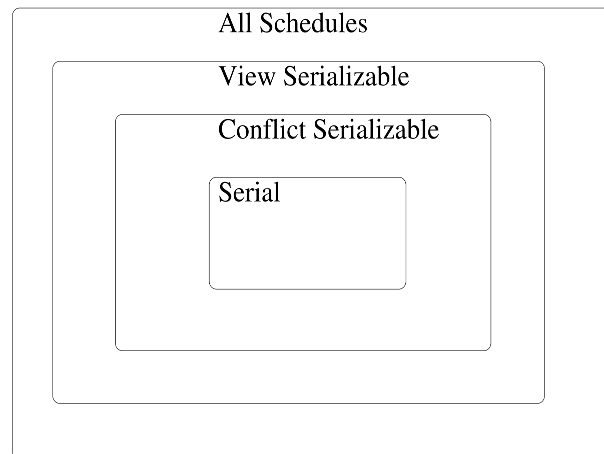

Figure 5.1: Venn Diagram for Classes of Schedules

We explore the basic locking method to ensure the One-copy Serializability among the update transactions and maintain the same current degree of read by the control matrix. Each update transaction makes a new version of data on the server and the server will broadcast the current version of data to client in each broadcast cycle. Thus client can read the correct and current data from the broadcast channel without communicating with the server and also improve the performance of the model. For example,

H_0: $r_1(x)w_2(y)r_1(y)w_2(x)w_1(z)c_1c_2$

Since $r_1(x)<w_2(x)$ and $w_2(y)<r_1(y)$, SG(H_0) consists of the cycle $T_1 \longrightarrow T_2 \longrightarrow T_1$. Thus, H_0 is not conflict serializability. In fact, this history can be accepted by our

model, since all transactions have the same read from relation, that is, they read the same version data from the server.

However, H_1: $r_1(x)w_2(y)w_1(z)w_2(x)c_2r_1(y)c_1$

H_1 is not accepted by the conflict serializability nor our model, since t_1 first read x from certain transaction, and then read y, which has been already updated by t_2. It will be checked by the control matrix and aborted since it can't meet the read condition.

5.2.2 *APPROX* – Update with Broadcast Disks in [SNSR99]

Server In *APPROX* model, there are three responsibilities for the server: broadcasting the committed and current version data to client, ensuring the consistency of the data in server, and updating and transmitting the control matrix with the data being broadcast.

Broadcast One broadcast cycle is defined as the completion of total disks broadcasting all data objects. For each broadcast cycle the server will broadcast the latest broadcast version of the data objects to client at the beginning of the cycle. There are two versions of data in the server: the latest committed version and the latest broadcast version. Before the beginning of each broadcast cycle, the server will update the latest version data with the latest committed version.

Optimistic Methods for Concurrency Control *APPROX* maintains the concurrency control among the transactions by the optimistic method proposed in [KR81], which is a non-locking and timestamp concurrency control mechanism.

The optimistic method divides the transaction into three phases: a read phase; a validation phase and a write phase. In a read phase, all readings are allowed and all writings will be updated locally, and copies will be made besides recording the timestamp at the beginning and the ending of the read phase. After the transaction with its data set are sent to the server and has passed the validation during its validation phase, the local copies of the transaction can be made global in its write phase. The server determines the validation of the transaction by comparing their timestamp at the beginning and the ending of their phases.

Control matrix If there are n data objects in the database, a matrix of size $n \times n$ will be used. In each broadcast cycle, the server will broadcast the control matrix together with the latest broadcast data objects. After a whole broadcast cycle, the server will update the control matrix entry and the data being broadcast from the latest committed space by the committed transactions.

Initially, each entry of the control matrix C is set to the cycle number 0. Assuming C_{new} and C_{old} are the control matrix with respect to the cycle number c_{new} and c_{old}, let $c_{new} > c_{old}$ be the cycle in which a newly transaction t commits. When transaction t updates both data object i and j, and the last transaction is to update object j, the entry $C_{new}(i,\ j)$ is set to the cycle in which t commits. If t only updates object j, then it depends on the set of all transactions that directly or indirectly updates the latest values of data objects in the read set of t. Thus the entry of $C_{new}(i,\ j)$ is set to the maximum cycle of $C_{old}(i,\ k)$, in which object k is the data in the read set of t. Lastly, if an object is not updated, the entry will then

be $C_{new}(i, j)$=$C_{old}(i, j)$.

The potential disadvantage of the control matrix is that the bandwidth required transmitting during each broadcast cycle. The size of control matrix is $n^2 \times \log(n)$, where n is the number of objects in the database. This overhead will be large for large value of n. In each broadcast cycle, if the size of each data object is s, the fraction of transmitting for control matrix is $\frac{n^2 \times \log(n)}{n^2 \times \log(n)+s}$. Then the overhead of the control matrix is not large when the size of data object is large. According to the above result we assume that the size of data object is quite large and then the overhead of transmission of the control matrix can be looked very small in our simulation experiments.

Client There are four operations performed in the transaction of client: read, write, commit and abort. The client retrieves the broadcast data page from the broadcast channel for operating transaction and sends the request and commit message to the server.

Read Any client who wants to perform a read-request should check the control matrix with the data to determine whether the read-operation can be performed. For a transaction t at the client, a read-operation on ob_j is allowed to execute in a cycle iff

$$\forall (ob_i, cycle) \in \mathrm{R}_t \; (\; C(i, j) < cycle \;),$$

where R_t is the set of $(ob_i, cycle)$ pairs such that transaction t previously read object ob_i from broadcast cycle *cycle* and C is the control matrix of the current broadcast cycle. If the condition failed, then this read-operation is aborted and the whole

transaction will be aborted and restarted.

Write The update operation can be performed locally without any check.

Commit After the whole transaction is completed and if the transaction requires no update operation, then the transaction will be committed. Otherwise, the client will send a committed message with the list of all updated objects and respective values to the server. In order to update the control matrix the client also send the set of objects read in the transaction to the server to check for consistency. The server will then send the commit or abort message back to the client. After the client has received the commit message, a new transaction will be generated.

Abort If a transaction is a read-only, then the abort will only restart it. If the transaction has performed the update operation, upon receiving the abort message from the server, it will clear the copies that need to be updated locally and will then be restarted at the client.

5.2.3 Basic Locking Update with Broadcast Disks

Server In our model, there are also three responsibilities for the server: broadcasting the committed and the current version of data to client; ensuring the consistency of the data in server and updating; and transmitting the control matrix with the data that is being broadcast.

Broadcast The broadcast functionality is the same as that described in Section 5.2.2. Based on the idea of the multi-disk broadcast in [AAFZ95], we make

an improvement in the sorting of all the data objects by its accessed probability. Hence, greater probability means that the object will be accessed more frequently. On the contrary, smaller probability means the object will be accessed less often. The objects will then be allocated to different disks with different speeds according to their probability. The objects with greater probability will be allocated in the faster disks and the objects with smaller probability will be set to the slower disks.

Basic locking concurrency control In order to ensure the consistency of the database, the server employs the basic locking concurrency control mechanism. When the server receives a request for updating a data object from the clients, it will first check the locking table as to whether the data is locked by another client. If this data object is not locked, the server will send the response message to the client to process the transaction. If locked, the server will check whether there is a deadlock between the waiting transaction and the owner transaction. If there is no deadlock, this client has to wait for the release of this data and will then enter a queue, otherwise, the server will send an abort message to the client and release all locks the transaction has obtained. When the server receives the committing message from the client, it will release all update locks and write the new version of data into the latest committed version space.

Control matrix The functionality and update of the control matrix are also like that in Section 5.2.2. Details of the server algorithm in our model is shown as Figure 5.2.

Algorithm

```
1.  page = (TRANSACTION *) get((HEAD *) broadcastchannel);
2.  switch(page.tag){
3.        case 'p': /*read page*/
4.              NewBroadcast(page); /*new broadcast cycle*/
5.              Break;
6.        case 'r': /*request from client*/
7.              If(Lock(page)==Ture) /*obtaining the write lock*/
8.                    SendResponse(page.client_id) /*Send the response to client*/;
9.              Else If(Deadlock(page)==Ture); /* Check the deadlock*/
10.                         Abort(page.client_id); /* Abort the transaction for deadlock*/
11.                   Else EnterQueue(page.client_id); /*Wait for lock in queue*/
12.             Break;
13.       case 'c': /*commit message from client*/
14.             Unlock(page.client_id); /*release write lock*/
15.             Update(page) /*update control matrix*/
16.             Break;
17.       case 'a': /*abort message from client*/
18.             Abort(page.client_id); /*Abort the transaction*/
19.             Break;
20. }
```

Figure 5.2: Server Loop for Our Model

Client There are also four operations performed in the transaction of client in our model: read, write, commit and abort. The client retrieves the broadcast data page and response message from the broadcast channel for operating transaction and sends the request and commit message to the server. Details of the client algorithm is shown as Figure 5.3.

Read Like the read functionality in Section 5.2.2, Any client has to check the control matrix with the data for processing the read-operation by the same condition in Section 5.2.2.

Write For an update operation at the client, the client will first send a request to server for obtaining the write-lock on this data object and wait until receiving the

Algorithm

```
1.  While(Transcouter < TRANSLENGTH){
2.        Newop(Transcouter); /*Process a new operation*/
3.        If(my.kind=='w')
4.              Send(page);/*Send request page for write lock*/
5.        While(1 > 0) { /*monitoring loop*/
              /*get the page from the broadcast channel*/
6.              page = (TRANSACTION *) get((HEAD *) broadcastchannel);
7.              If(my.kind=='w' && page.destination == my.client_id){
              /*waiting for write response from server*/
8.                    switch(page.tag){
9.                          case 'r': /*obtain the write lock*/
10.                               Update(page); /*update locally*/
11.                               Transcouter++;
12.                               break;
13.                         case 'a': /*abort message*/
14.                               Abort(); /*abort the local transaction*/
15.                               break;
16.                   }
17.                   break;
18.             }
19.             Else /*Broadcast reading*/
20.                   If(my.data_id == page.data_id && page.tag == 'p' && my.kind=='r'){
21.                         If(CheckMatrix(my.data_id)==True) /*Check the control matrix for read*/
22.                               Transcouter++;
23.                         Else
24.                               Abort(); /*Abort the local transaction */
25.                         Send(page);/*place page back to the channel*/
26.                         break;
27.                   }
28.             Send(page);/*place page back to the channel*/
29.       }
30. }
31. Commit(my.transaction_id); /*Commit the transaction to server*/
```

Figure 5.3: Client's Algorithm in Our Model without Cache

guarantee or the abort response from the server. The updated set of this transaction will then be sent to the server to detect the deadlock on the server. If the client has obtained the write lock, it will retrieve the response page, update it locally and process the transaction. Otherwise, this transaction will be aborted.

Commit After the whole transaction is finished, the client will send a committed message with the list of all updated objects and respective values to the server, which will commit this transaction and release all write-locks of this transaction. In order to update the control matrix, the client will also send the set of objects to be read in the transaction to the server.

Abort If a transaction is a read-only, then the abort will only restart it. If the transaction has obtained the writing lock on the server, it should send an abort message to the server to release all the locks obtained and will then be restarted at the client.

5.2.4 Discussion

Table 5.1 shows the differences between our model and that in [SNSR99]. Details of the correctness criteria has been discussed in Section 5.2.1. In our model the deadlock problem is issued by two update data sets of one transaction, in which one of the set of data has obtained lock and the other set of data is to obtain lock. When the client wants to obtain the lock of one data object and the data object is locked by another client, this client will have to check whether its obtained set overlaps the set to obtain of the transactions in the waiting queue. If there is no overlap, it will

Model	*APPROX*	Our model
Broadcast	Broadcasting data with control matrix	
Concurrency Control	Optimistic concurrency control	Basic locking mechanism
Control matrix	Computing control matrix in Section 5.2.2	
Read at client	Only check the read condition in Section 5.2.2	
Write at client	Write locally and after validation then make global	Request the write lock from server and when commit or abort it will be released
Commit	Commit to server with read and write set to check the consistency	Send write set to server and release all locks obtained
Abort	Restart the transaction and clear the local copy	Restart the transaction and release all locks obtained
Disadvantage	Maintain the identity of the global time	Detecting the deadlock

Table 5.1: Comparison of *APPROX* and our model

then enter the queue, in which otherwise, it will then be aborted. The server will subsequently send the abort message to client.

5.3 Update with Cache in Client

Currently, the local memory is utilized in each client to cache the objects being broadcast so as to improve the model performance. Problems on data consistency arise when client caching is used. If a client transaction is committed on the server, then the data in the update set must be effected. Moreover, replications of these

data objects in the client memory should be affected. Data inconsistency will occur when another transaction access these data objects. In order to keep the clients' cache consistent with the update value on the server, the affected replications in the clients' cache must be invalidated.

5.3.1 Server

In our model, the control matrix is not only used in the concurrency control but also can be an invalidation report, which is used to make the stale replication invalidated in the client memory. The control matrix records the objects, which will be update by certain transaction by the method presented in previous Section 5.2.3. Until the server has committed the transaction for a client, the new version of control matrix will be modified and broadcast with the new version data, so as to notify all clients that some replications in the cache should be invalidated in new broadcast cycle. Thus there is no difference on the functionality of the server between the model without cache and with cache. Detail of the server algorithm is shown as Figure 5.2.

5.3.2 Client

Besides the functionalities in Section 5.2.3, there are some modifications on the read- and write-operation for cache management in the client of this model. Detail of the client algorithm is shown as Figure 5.4.

Read For a read request of a client transaction it will first check whether the preferred data object has been cached in the local memory. If the preferred data is not in the memory, the client will monitor the broadcast channel and wait for the preferred data object. After consistency is checked for read operation by the

Algorithm

```
1.  While(Transcouter < TRANSLENGTH){
2.        Newop(Transcouter); /*Process a new operation*/
3.        If(my.kind=='w')
4.              Send(page);/*Send request page for write lock*/
5.        If(my.kind=='r'){ /*the needed data is in cache*/
6.              CheckCache(); /*invalidation of the cache*/
7.              If(InCache(my.data_id)==True && CheckMatrix(my.data_id)==Ture)
8.                    Transcouter++;
9.              continue;
10.       }
11.       While(1 > 0) { /*monitoring loop*/
              /*get the page from the broadcast channel*/
12.             page = (TRANSACTION *) get((HEAD *) broadcastchannel);
13.             If(my.kind=='w' && page.destination == my.client_id){
                /*waiting for write response from server*/
14.                   switch(page.tag){
15.                         case 'r': /*obtain the write lock*/
16.                               Update(page); /*update locally*/
17.                               Transcouter++;
18.                               break;
19.                         case 'a': /*abort message*/
20.                               Abort(); /*abort the local transaction*/
21.                               break;
22.                   }
23.                   Send(page);/*place page back to the channel*/
24.                   break;
25.             }
26.             Else /*Broadcast reading*/
27.                   If(my.data_id == page.data_id && page.tag == 'p' && my.kind=='r'){
28.                         If(CheckMatrix(my.data_id)==True) /*Check the control matrix for read*/
29.                               Transcouter++;
30.                               RefreshCache(my.data_id); /*Refresh the local memory for read*/
31.                         Else
32.                               Abort(); /*Abort the local transaction */
33.                         Send(page);/*place page back to the channel*/
34.                         break;
35.                   }
36.             Send(page);/*place page back to the channel*/
37.       }
38. }
39. Commit(my.client_id); /*Commit the transaction to server*/
40. Refresh(my.wset);/*Refresh local memory for committed write set*/
```

Figure 5.4: Client's Algorithm in Our Model with Cache

read condition in Section 5.2.3, the client will cache them into local memory and record the respective entry of the data cached in the control matrix. For example, if ob_j is cached, then *C(j, j)* will be recorded. If the preferred data is in the local memory, the client will retrieve the control matrix with a broadcast data object from the broadcast channel before checking of consistency. This is to check whether the preferred data object to read is validated, that is, the client will compare the same entry between that in the matrix and in the cache. When a client finds invalidated data objects in the local memory by the control matrix, these replications will then be removed from the memory regardless how recent or how often they have been accessed, and wait for the preferred data that is being broadcast in the channel.

Write For any update operation, the client will send the request to the server for the write lock on the data object and write locally after receiving the response message that is sent from the server. After the transaction is committed to the server, the client will refresh the memory by certain cache management policy that is in accordance to the write set of the committed transaction.

Cache manage policy When the local cache is full, a victim must be chosen from the local memory to replace the new data object. According to the broadcast environment, we choose two caching manage policies to compare the performance of our model, one is *LRU* (Least Recently Used) [AAFZ95] algorithm, other is *LIX* in [AFZ96b]. *LRU* maintains the cache as a list. When an object in the cache is accessed, it is moved to the top of its own list. When a new object enters the cache, *LRU* will choose the object that is on the bottom of the list as the victim, and the

new object will be inserted to the list. Like *LRU*, *LIX* is an efficient constant time approximation of *PIX*. *LIX* also maintains the chains in each client memory by the order of the probability of access, but it maintains one queue for each disk. When the cache is full and a new object must enter, *LIX* will evaluate the *lix* value of the objects at the bottom of each chain and evict the object with the lowest *lix* value. The new object will then enter the chain corresponding to the disk it is on. The *lix* value of each object is computed by the probability estimate for an object divided by its broadcast frequency.

5.4 Summary

Based on the result of theoretical analysis and experimentation mentioned in the previous chapters, we have constituted a sound basis for the implementation of concurrency control strategies for the interleaved execution of transactions. In this chapter, we have devised and studied the lock-based concurrency control strategies for transactions in the broadcast disks. We proposed that the concurrency control mechanism be combined by a weaker serializability, called One-copy serializability, with the lock-based strategies. The One-copy serializability will be checked by the control matrix in [SNSR99] and the write-write conflict will be checked by the lock-based strategy in the server. Furthermore, we explored the different improvements of the performance of broadcast disks model with cache in client and utilized the control matrix to solve the problem of the inconsistent cache. In the next chapter we will evaluate the performance of these models and compare the performance of two different caching policies in the broadcast disks by simulation experiments.

Chapter 6

Performance Analysis for Database Transaction with Locking in Broadcast Disks

We empirically evaluate the performance of the model we propose. For the purpose of the performance evaluation, we build a simulation using the CSIM [JH99] discrete event simulator. The primary performance metrics we use is the response time in which one committed transaction. The lower the response time, the better the performance of our model.

6.1 Experimental Set-up

The simulator measures performance in logical time units, which equals to the time required to broadcast a single object, request, or message. In the sequel, we measure the response time within 1000 client transactions and the results are derived from the last 500 transactions so as to ensure that the results form a steady-state model.

Our simulator consists of a server, some clients and a broadcast channel. The data objects accessed by transaction follows a Zipfian distribution[AAFZ95] in the database. The parameter ρ indicates the ratio of read operation over update opera-

Item Number	30,60,90,120,150
Client Number	64
Client Transaction Length	4
θ	1, 0.8, 0.6, 0.4, 0.2, 0
ρ	1, 2, 4, 8, 16
CacheSize	5

Table 6.1: Parameter of the experiments on transaction process

tion of one transaction for any client. The parameter θ controls the probability for any object to be requested in any transaction (either read- or write-request following the previous parameter ρ). θ is the parameter of a Zipfian distribution of the form: $p_i = (1/i)^{\theta}$. A value of zero for θ indicates a uniform distribution while a value of one indicates a skewed distribution (few objects are very frequently accessed). Client transaction length indicates the number of read/update operations in one transaction at the client and each transaction has a commit or abort operation as its end. The successive values used in the experiments are reported in Table 6.1.

6.2 Comparative Performance Analysis

6.2.1 Updates in Clients with Different Concurrency Control Mechanism

In this experiment, all update operations generate only in the client transaction, not in server transaction. We compared the experiment results between *APPROX* and our model with different ratio of read-operation to write-operation.

In Figure 6.1 we find that the performance of both models is improved with an increased of the ratio of read- to write-operation. Similar to our previous research work, the broadcast channel can make more improvements on the model by increasing the number of read-operation. That is, the more read request from client,

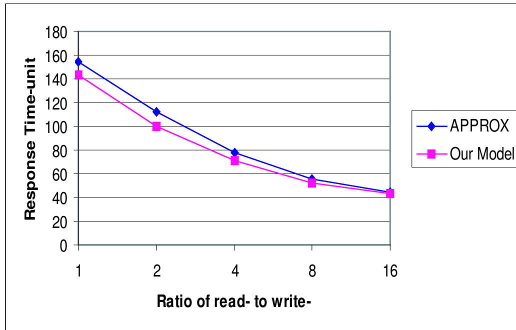

Figure 6.1: Response time at θ=1 and client=64, with varying ρ

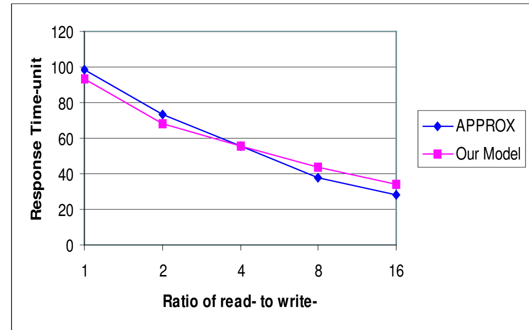

Figure 6.2: Response time at θ=0.5 and client=64, with varying ρ

the better performance the model has. On the other hand, we find that the degree of improvement of the two models is different. Although our model outperforms the *APPROX*, the improvement degree of our model is smaller than that of the *APPROX*. The reason for the difference is in the basic premise of Optimistic concurrency control in *APPROX* in which most transactions will not conflict. Hence, when the number of conflicts increased, its performance is worse than that of the basic locking. Thus more transaction will be accepted in our model.

However, when there are few conflicts among the transactions, that is, the probability of access data become uniform, Figure 6.2 shows that the performance of both models has improved with increasing of the ratio of read- to write- operation. Furthermore, when the number of read operation becomes greater, the performance of the *APPROX* is better than that of our model. The reason for that is optimistic concurrency control can outperform the basic lock mechanism when there are few conflicts among the transactions.

Shown as Figure 6.3 we just find that the performance of both models will be decreased by the increasing of the number of data object. The main reason for that

is the increasing of the number of data object makes the length of each broadcast cycle longer. The longer broadcast cycle will make more transactions to commit to the server and more possible conflicts. Therefore, the response times increase with the number of data object.

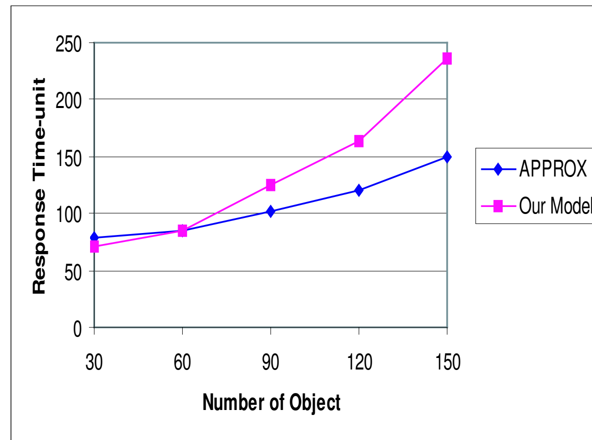

Figure 6.3: Response time at θ=1, ρ=4 and client=64, with varying number of data object

6.2.2 Updates in Clients with Broadcast Disks Environment

In previous experiment, the broadcast disks environment is a flat broadcast disk environment, in which a data object will be broadcast in the same frequency in spite of their different accessed probability. In order to improve the performance of the model, multi broadcast disks in [AAFZ95] is utilized in this model. We choose 30 data objects in the server and 64 clients in the model. The length of transaction in client is 4. And the relative speed of the multi disks is 5: 3: 1.

Shown in Figure 6.4 and 6.5, we can find that, within the different situations in which the data objects accessed by client are being accessed more frequently and less concentrative, the performance of multi-disk broadcast will be better than that of the flat broadcast disk when the ratio of read- to write- operation increased. The

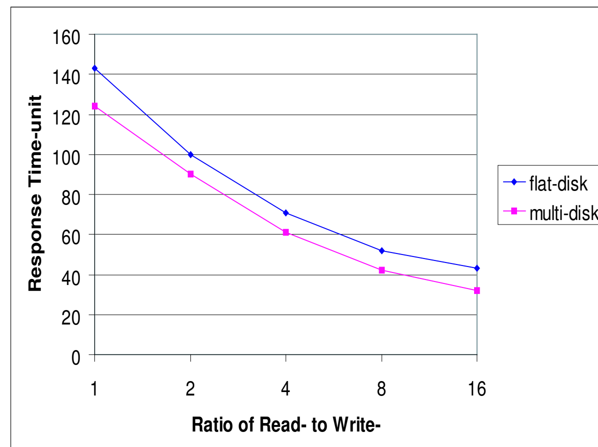

Figure 6.4: Response time at θ=1 and client=64, with varying ρ

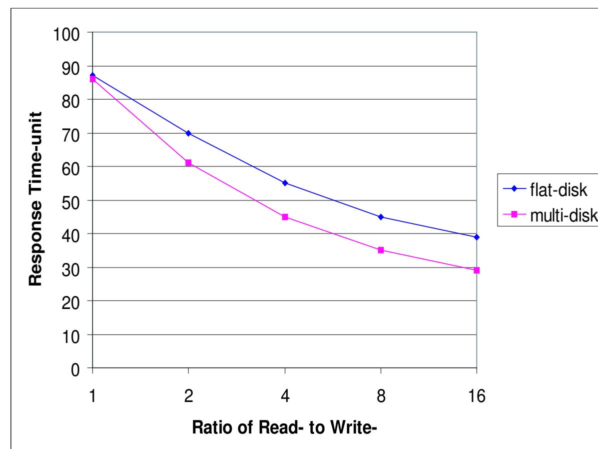

Figure 6.5: Response time at θ=0.5 and client=64, with varying ρ

multi-disk broadcast environment has two advantages. One being the time in waiting for preferred data objects in the broadcast channel shortens in the multi-disk broadcast environment, as the hotter data will be broadcast more often than the colder data. The other is that the client will spend less time finishing the transaction locally in multi-disk broadcast environment. Thus the number of broadcast circle for completing the whole transaction locally to commit is less than that in flat broadcast disks environment. In this model we also utilize the control informa-

tion matrix like in [SNSR99]. Thus these advantages can make more read-requests and committed transactions and improve the performance of model in multi-disk broadcast environment.

6.2.3 Updates in Clients with Broadcast Disks Environment with Cache in Client

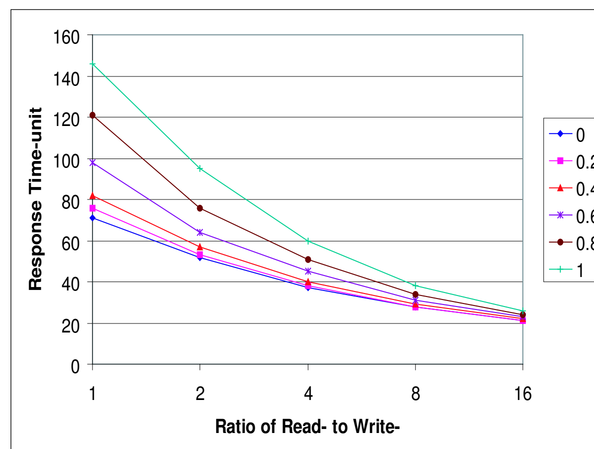

Figure 6.6: *LRU*: Response time at client=64, with varying ρ for different θ

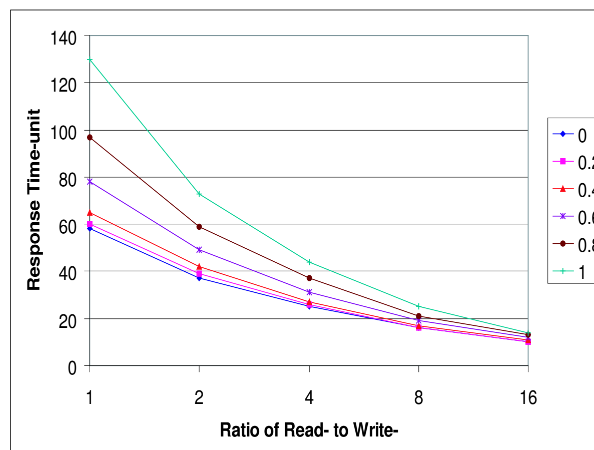

Figure 6.7: *LIX*: Response time at client=64, with varying ρ for different θ

Now in multi-disk broadcast environment, we explored the local memory in the client to cache some data objects so as to improve the the performance of the model.

We used *LRU* and *LIX* cache policy to study the similarity and difference between them. First, we investigate the change of the performance of the model by varying the ratio of read- to write-operation and the distribution of accessing data objects. In Figure 6.6 and Figure 6.7, when the parameter ρ becomes greater, that is, there are more read operations in the transaction, the performance of the model has shown to improve with respect to the different accessed probability distributions in the model used the *LRU* and *LIX* cache policy. In fact, the number of update operation will affect the performance of the model more. The first reason is because the transaction, including update operation, has to be sent to the server for committing, and the second is that the server will have to modify the data object updated by the committed transaction and the control matrix in order to broadcast to the clients. These two reasons will cause the transaction to be aborted and restarted due to the conflicts among the transactions.

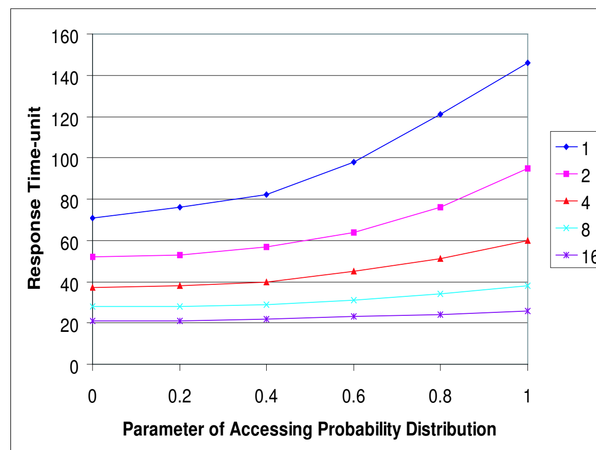

Figure 6.8: *LRU*: Response time at client=64, with varying θ for different ρ

Next, we study the effects of the accessing probability on the performance of these two models. Figure 6.8 and Figure 6.9 showed that with an increased of

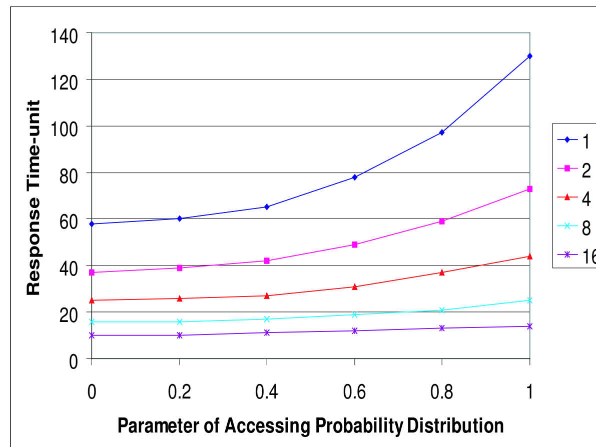

Figure 6.9: *LIX*: Response time at client=64, with varying θ for different ρ

the parameter of the accessing probability distribution, which means that the data objects accessed become more concentrative, the performance of the model will decrease. The main reason for this decrease is in the conflicts of writing the concentrative data objects. When the parameter of the distribution becomes greater, there will be more concurrency conflicts for the same data objects in the model due to the competition for hotter data objects.

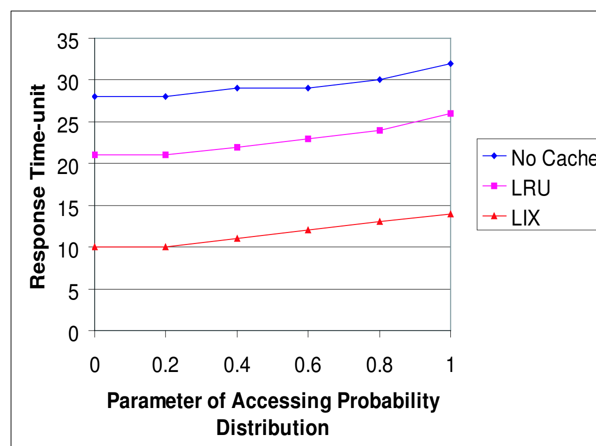

Figure 6.10: Response time at client=64 and ρ=16, with varying θ

Furthermore, we compare the performance among the models without cache

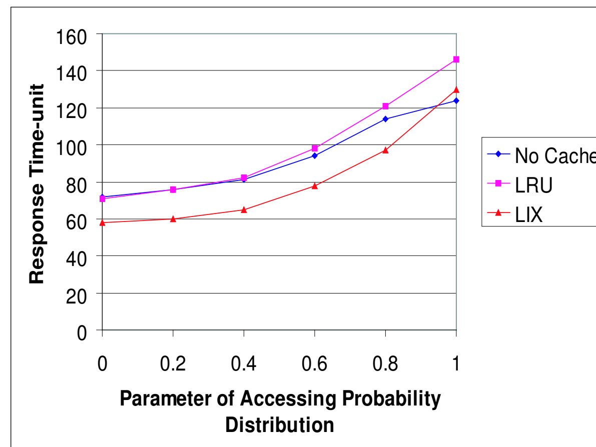

Figure 6.11: Response time at client=64 and ρ=1, with varying θ

and with cache in clients. Shown in Figure 6.10, the performance of the model with cache is better than those of the models without cache. Although the invalidation of the cache will cost some system time, the improvement of the retrieving from the local memory is greater than the invalidation cost when the ratio of read- to write- operation is 16, that is, there are more read-operations. However, all the performance of the models will decrease as the data objects updated become more concentrative. In Figure 6.11, when the ratio of read- to write- operation equals to 1, the performance of the model without cache and the model with LRU are almost same. As the data objects updated becoming more concentrative, the performance of LRU and LIX, which normally more adapts to the multi-disk broadcast environment and are better than that of the no-cache model, are not better than that of no-cache model. The cost of invalidation for client reduces the advantage of LIX and LRU, which is the strategy of choosing the victim, even though it will keep the hotter data object to be updated in the local memory. Thus the performances of the models with cache have been reduced.

6.3 Summary

We proposed for the models on transactions in the broadcast channel based on the basic locking model. A control matrix is explored for maintaining the consistency of the read-operation of a transaction, as update-operation of a locking mechanism is responsible for maintaining the consistency of the whole database. We have compared the performance of the APPROX and our model and found that the basic lock concurrency control outperforms the optimal concurrency control when there are more conflicts in the model. Moreover, we utilized the broadcast disks mechanism in order to improve the performance of the broadcast model, in which we allocated all the data objects into different disks with different speeds. Hence, the multi-disk mechanism will improve the types of performance in the broadcast environment. Furthermore, we investigated the different improvement for the performance of broadcast disks model with cache in client, and compared performance of the two different caching policies utilized by the broadcast disks. As a result, the broadcast mechanism is a viable mechanism for the database system accepting the update while maintaining the concurrency control of the transaction processing.

Chapter 7

Conclusions and Future work

In this chapter, we will conclude this thesis and outline the future work.

7.1 Contributions

In this thesis we studied the design and evaluated the performance of update model controlled by a basic locking mechanism for broadcast disks architecture that involves replication on the broadcast channel and in the clients' caches.

- ◇ We have presented an update and locking mechanism for a broadcast disks model with simple operation and cache. The new concurrency control mechanism of this model outperforms the traditional client-server model due to combination of the advantages of the locking and the broadcast disks environment. There can now be, on the broadcast channel, several copies of the same object available for reading as well as one copy available for update by one identified client or/and exclusively one already updated copy, not available for reading or writing. Clients can read the previous version of an object while the current version is being updated. This proposal constitutes a sound basis for the implementation of concurrency control strategies for the interleaved

execution of transactions. The models we devised and studied can support the implementation of schedulers or lock-based concurrency control strategies.

◇ We have extended our research from simple data operation into transaction processing in the broadcast disks environment. In order to guarantee the correctness of the concurrency among the transactions, we have combined locking mechanism and multi-version concurrency control. Our proposed models based on broadcasting are efficient. They incrementally yield better performance by the basic locking mechanism when the opportunities for sharing objects for read are high: high ratio of read over write operations, large number of clients, and skewed distribution of popularity of objects. Thus, broadcasting remains a viable alternative even in the presence of updates by the client. The better performance of our models is particularly important since broadcasting is candidate architecture for many new applications involving new devices and networks.

In summary, our contributions are the proposal of a new concurrency control mechanism combined by the basic locking and the broadcast disks in the peer-to-peer environment. From the results and analysis of our experiments, we have validated and illustrated the efficiency of our concurrency control mechanism, thus paving the way for the future work in this environment.

7.2 Future work

Broadcasting technology of data dissemination is very new in comparison with other technologies. When we implement this model using the locking concurrency control

mechanism, it is not too complicated for the concurrency control between the server and the client with local memory. Nevertheless, there is a greater need to modify existing protocols for better performance in the broadcast disks environment. The modifications should be easy to implement and add on to the original protocol with minimal modification to the original model.

In our future work, based on our previous research work we could study the recovery mechanism in the broadcast disks environment and investigate the optimistic broadcast schedule method on the server.

We will focus on the optimistic broadcast schedule of the broadcast content to enhance our system with reinforcement learning ability in the multi-disk broadcast. A reinforcement model should include some elements [LPKM96]: a policy mapping from state to action; a reword function and a value function indicating what is good in an immediate sense and in the long run respectively. Within a reinforcement model, the server plays an agent role. It could sense the actual need of the client as states of the model and adaptively broadcast data by learning from its prior mistake and experiences. We will investigate the appropriate reinforcement model to improve the performance of the broadcast disks model.

Bibliography

[AAFZ95] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. Broadcast disks: data management for asymmetric communication environments. In *ACM SIGMOD Intl. Conference on Management of Data (SIGMOD 95), San Jose, CA*, pages 199–210, 1995.

[AFZ96a] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Disseminating updates on broadcast disks. In *The VLDB Journal*, pages 354–365, 1996.

[AFZ96b] Swarup Acharya, Michael J. Franklin, and Stanley B. Zdonik. Prefetching from broadcast disks. In *ICDE*, pages 276–285, 1996.

[AFZ97] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *ACMSIGMOD 97*, pages 183–194, 1997.

[BC92] P. M. Bober and M. J. Carey. Multiversion query locking. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, 1992.

[Cao02] G. Cao. On improving the performance of cache invalidation in mobile environments. *ACM/Kluwer Mobile Networks and Application*

(MONET), 7(4):291–303, 2002.

[FCL97] Michael J. Franklin, Michael J. Carey, and Miron Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, 22(3):315–363, 1997.

[GHW87] K. Lee G. Herman, G. Gopal and A. Weinrib. The datacycle architecture for very high throughput database systems. In *Proc. of ACM SIGMOD, San Francisco, CA, May*, 1987.

[JH99] P. Herout J. Hlavicka, S. Racek. C-sim v.4.1. In *Research Report DC-99-09, Sep*, 1999.

[JMR98] H. V. Jagadish, Inderpal Singh Mumick, and Michael Rabinovich. Asynchronous version advancement in a distributed three-version database. In *ICDE*, pages 424–435, 1998.

[KR81] H.T. Kung and John T. Robinson. On optimistic methods of concurrency control. In *TODS*, pages 213–226, 1981.

[LCC99] K.-Y. Lam, E. Chan, and J. Chun-Hung Yuen. Broadcast strategies to maintain cached data for mobile computing system. *Lecture Notes in Computer Science*, 1552:193–204, 1999.

[LKTL00] Kam-Yiu Lam, Tei-Wei Kuo, Wai-Hung Tsang, and Gary C. K. Law. Concurrency control in mobile distributed real-time database systems. *Information Systems*, 25(3):261–286, 2000.

[LPKM96] M. L. Littman L. P. Kaelbling and A. W. Moore. Reinforcement learning: A survey. *Journal of Artifical Intelligence Research*, (4):237–285, 1996.

[MT99] P. Valduruez M. Tamer. *Princles of Distributed Database System.* Prentice-Hall, second edition, 1999.

[PAB00] Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems.* Microsoft, 2000.

[RR00] J.Gehrke R. Ramakrishnan. *Database Management System.* McGraw-Hill, second edition, 2000.

[SL99] HeongChang Yu SangKeun Lee, Chong-Sun Hwang. Supporting transactional cache consistency in mobile database systems. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access, August 20*, 1999.

[SNSR99] Jayavel Shanmugasundaram, Arvind Nithrakashyap, Rajendran Sivasankaran, and Krithi Ramamritham. Efficient concurrency control for broadcast environments. pages 85–96, 1999.

[TFBW92] G. Herman T. Hickey K. C. Lee W. H. Mansfield J. Raitz T. F. Bowen, G. Gopal and A. Weinrib. The datacycle architecture. In *Communications of the ACM, December 1992, Vol. 35, No. 12.*, 1992.

[Tho98] A. Thomasian. Concurrency control: Methods, performance, and analysis. In *ACM Computing Surveys*, pages 70–119, 1998.

[VI95] S. Viswanathan and T. Imielinski. Pyramid broadcasting for video-on-demand service. In *Proceedings of the SPIE Multimedia Computing and Networking Conference, San Jose, CA, February*, 1995.

[WN90] K. Wilkinson and M. A. Neimat. Maintaining consistency of client-cached data. In *Proceedings of the Sixteenth International Conferenceon-Very Large Databases*, 1990.

[YH01a] Y. H. Lee Y. Huang. Caching broadcasted data for soft real-time transactions. In *Proceedings of the 5th IIIS SCI/ISAS Conference, Orlando, Florida, July*, 2001.

[YH01b] Y. H. Lee Y. Huang. Stubcast - efficient support for concurrency control in broadcast-based asymmetric communication environment. In *Proceedings of the 10th IEEE ICCCN Conference, Scottsdale, Arizona, October*, 2001.

[ZGE01] Y. Zhao, R. Govindan, and D. Estrin. Residual energy scans for monitoring wireless sensor networks. In *Technical Report 01-745, May*, 2001.