Broadcast Disks:
Dissemination-based Data Management
for
Asymmetric Communication Environments

by
Swarup Acharya
B. Tech. (Hons)., Indian Institute of Technology, Kharagpur, 1991
Sc. M., Brown University, 1993

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1998

This dissertation by Swarup Acharya
is accepted in its present form
by the Department of Computer Science
as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ________________________________
Stanley B. Zdonik

Recommended to the Graduate Council

Date ______________ ________________________________
Thomas W. Doeppner

Date ______________ ________________________________
Maurice P. Herlihy

Date ______________ ________________________________
Michael J. Franklin
University of Maryland, College Park

Approved by the Graduate Council

Date ______________ ________________________________

Vita

Swarup Acharya was born on September 23rd, 1969, in Calcutta, India. He did primary and secondary schooling in various cities in India and received his Secondary (Class X) and Higher Secondary degrees (Class XII) from Kendriya Vidyalaya, Air Force Station in Jamnagar, Gujrat. In August 1987, he joined the Indian Institute of Technology, Kharagpur for his baccalaureate studies. He was awarded a Bachelor of Technology (Honours) degree in Computer Science and Engineering in May 1991. Subsequently, he joined the doctoral program in the Computer Science Department at Brown University, Providence and he received an Sc.M degree in Computer Science in May 1993.

Abstract

The increasing ability to interconnect computers through internetworking, wireless networks, high-bandwidth satellite, and cable networks has spawned a new class of information-centered applications based on *data dissemination*. These applications, which often serve huge client populations, employ broadcast to efficiently deliver data to the clients. In data dissemination, the data transfer is initiated by the server, inverting the traditional relationship between the client and the server. This thesis proposes a novel "multi-disk" framework for data dissemination called ***Broadcast Disks***. The Broadcast Disks approach significantly improves upon the previous work in dissemination-based systems and raises a number of fundamentally new research challenges.

In this thesis, we first motivate why the rise of *asymmetric* environments (i.e., networks which have a significantly higher bandwidth available from servers to clients than in the reverse direction) and the scale of the emerging distributed information systems is causing a shift from the traditional *pull-based* client-server model to a *push-based* dissemination model. Then, the Broadcast Disks model is introduced and explored using a simulation-based study and a working implementation.

The bulk of this thesis uses simulation results to understand the basic tradeoffs in a dissemination framework. We demonstrate the performance benefits of the Broadcast Disks model over the traditional approach to structuring a dissemination program. We also introduce a new *cost-based* approach to client cache management and develop efficient algorithms for prefetching and dissemination of updates. Then, the thesis addresses the issues that arise in supporting clients using both push and pull-based data delivery. Finally, we describe our experience in building and testing a Broadcast Disks prototype. While the studies using the prototype validate the simulation-based intuitions, they also raise many new issues and highlight some of the shortcomings of the current technology for building push-based systems. The underlying theme driving the studies in this thesis is to develop techniques to improve system performance and scalability which adapt to the new tradeoffs in the emerging computing landscape.

Acknowledgements

I would like to express my deepest gratitude to my advisor Prof. Stan Zdonik for his constant support and his great technical advice. Stan has been extremely encouraging throughout and right from the outset treated me more as a colleague than an advisee. His insights during our meetings would invariably bring clarity and very often open up new directions which I would have missed altogether. I enjoyed my grad tenure much more than I anticipated starting out and I owe a large part of it to Stan. I have also been very fortunate to have worked with Prof. Mike Franklin from University of Maryland. Mike was never more than a phone call or an email away even at crunch times. He has an amazing ability to convert globs of text (very often written by me) into an exquisitely organized paper in a single overnight session; an ability I hope I can halfway emulate someday. Mike has been a constant source of sage wisdom and I deeply appreciate all his help and support. Most of the research presented here is joint work with Stan and Mike and I thank them allowing me to use it in the thesis. They make a great team (both for advising and for comedy!) and I hope many more students benefit from their co-guidance. Thanks also to Prof. Maurice Herlihy and Prof. Tom Doeppner for being on my committee and for their comments on the earlier drafts.

Many thanks are also due to Dr. Rafael Alonso under whom I worked at Matsushita Labs in the summer of 1993. Rafael was the catalyst who gave the project a jump start by getting all of us under the same roof in Princeton. He was also a co-author on the initial papers. Rafael has written recommendations for me on more than one occasion and as if that wasn't enough, he even offered me a job at the end of it all.

A number of people in the department made the daily trip to the CIT worthwhile. My officemates, Dimitris, Lee and Steven, let me get more than my fair share of the Sparcs in the office. Dimitris Michailidis deserves a special thanks for all the LaTeX help and for being a moving encyclopedia. He always has the right pointers for any information you will ever need, be it NP-completeness or SLR cameras. Thanks to Mary Andrade who often went out of her way to solve any bureaucratic hurdles and to Rahul Bose for tying up the loose ends on the prototype.

I have to also thank a number of friends whose company was invaluable during my stay in Providence. Thanks much to people I have roomed over the years — Lalit, Manish, Nari and Ambar and to Bharathi, Santosh, Priya, Srikanta, Kavita, Nidhiya, Alok, and Vishy who were always great fun to be with and who helped me preserve my sanity. In particular, Vishy Ramachandran and Narayanan Subramanian deserve special thanks for always being around when the going really got tough and for bringing out the caffeine addict in me. Alok Kumar has my gratitude for all the long distance jam sessions.

Finally, I would like to thank my family – my elder brother Arup and my parents Manjari and Amal Acharjyya, without whose love, support and encouragement I wouldn't be where I am today. Being in a related area, I have bounced many a research idea off Arup and more importantly, he has been a source of inspiration throughout. *Ma* and *Bapi*, by their own examples, taught us the value of education and always encouraged us to do our best. For their love, wisdom and all the cheer, I will forever be grateful. This thesis is dedicated to them.

Credits

Portions of the research described in this thesis were done in collaboration with Rafael Alonso, Michael Franklin and Stanley Zdonik and have already appeared in jointly authored papers. Chapter 5 and the initial sections of Chapter 3 are from [AAFZ95] which was co-authored with Rafael Alonso, Michael Franklin and Stanley Zdonik. The results in Chapters 6, 7 and 8 appeared in [AFZ96b], [AFZ96a] and [AFZ97] respectively and were all jointly authored with Michael Franklin and Stanley Zdonik.

Benjamin Boer, Rahul Bose, Jinggang Huang and Blossom Sumulong contributed to building the prototype described in Chapter 9 and Jie Cui helped generate some of the results in that chapter.

I was supported by various grants during my graduate career including ONR grant number N00014-91-J-4085 under ARPA order number 8220 and a gift from Intel Corporation. The support for the last two years (1995-97) came from a Co-operative Fellowship sponsored by IBM Corporation. I gratefully acknowledge this support.

Contents

Vita iii

Acknowledgements iv

Credits vi

Contents vii

List of Tables xi

List of Figures xii

1 Introduction 1
1.1 Background and Motivation 2
1.1.1 Client-Server Model 2
1.1.2 Issues in Client-Server DBMS Design 3
1.1.3 Data Delivery Choices 6
1.2 Contributions of the Thesis 9
1.2.1 Viability of Push-based Systems 9
1.2.2 Broadcast Disks : A Model for Periodic Broadcast 10
1.2.3 Cost-based Cache Management 10
1.2.4 Update Management and Cache Consistency 11
1.3 Organization of the Thesis 11

2 The Case for Push-based Systems 13
2.1 Recent Trends (or, Why a Change in Focus?) 13
2.1.1 Communication Asymmetry 13
2.1.2 Information-Feed Applications 16
2.1.3 Scale 17
2.2 Why Push? 18
2.2.1 Drawbacks of Pull Model 18

2.2.2 Advantages of Push . . . 20
2.3 Data Delivery Models . . . 21
2.3.1 Data Dissemination . . . 22
2.3.2 Requirements for Dissemination . . . 23

3 Broadcast Disks **26**
3.1 Introduction . . . 26
3.1.1 Broadcast Program Options . . . 27
3.2 Structuring the Broadcast Disks . . . 28
3.2.1 Properties of Periodic Broadcast Programs . . . 28
3.2.2 Broadcast Program Generation . . . 30
3.2.3 Dynamic Broadcast Programs . . . 33
3.2.4 Memory Hierarchy . . . 35
3.3 Client Cache Management . . . 37
3.3.1 Key Differences with Pull-based Systems . . . 38
3.3.2 Role of Caching in Broadcast Environments . . . 39
3.3.3 Cost-based Caching . . . 42
3.3.4 Prefetching . . . 43
3.4 Scope of the Thesis . . . 46
3.5 Approach of the Thesis . . . 48
3.6 Related Work in Data Dissemination . . . 49

4 Modeling a Broadcast Disks Environment **53**
4.1 Client Model . . . 54
4.1.1 Measured Client Execution Model . . . 54
4.1.2 Virtual Client Execution Model . . . 55
4.2 Server Model . . . 56
4.2.1 Server Execution Model . . . 56
4.2.2 Server Update Model . . . 59
4.2.3 Server Backchannel Model . . . 60
4.3 Experimental Methodology . . . 61

5 Demand-Driven Caching **63**
5.1 Introduction . . . 63
5.2 Performance of Cache-less Clients . . . 64
5.2.1 Experiment 1: No Caching, Basic Broadcast Tradeoffs . . . 64
5.2.2 Experiment 2: No Caching, Noise Sensitivity . . . 66
5.3 Performance of Standard Caching . . . 66
5.3.1 $\mathcal{P}$: A Standard Page Replacement Algorithm . . . 67

5.3.2 Experiment 3: Influence of Cache on Server Program . . . 67
5.3.3 Experiment 4: $\mathcal{P}$, Noise Sensitivity . . . 69
5.4 Performance of Cost-based Caching . . . 70
5.4.1 $\mathcal{PIX}$: A Cost-based Caching Algorithm . . . 70
5.4.2 Experiment 5: $\mathcal{PIX}$, Noise Sensitivity . . . 71
5.4.3 $\mathcal{LIX}$: Implementable Cost-based Caching . . . 72
5.4.4 Experiment 6: $\mathcal{LIX}$ vs. LRU . . . 74
5.5 Conclusions . . . 76

6 Prefetching from Broadcast Disks **78**
6.1 Introduction . . . 78
6.2 $\mathcal{PT}$: A Prefetching Cache Management Strategy . . . 78
6.2.1 Static vs. Dynamic Caching Metrics . . . 79
6.3 Experiments and Results . . . 81
6.3.1 Parameter Settings . . . 81
6.3.2 Experiment. 1: Uniform Access, Flat Disk . . . 82
6.3.3 Experiment 2: Skewed Access, Flat Disk . . . 84
6.3.4 Experiment 3: Uniform Access, Multi-Level Disk . . . 85
6.3.5 Experiment 4: Skewed Access, Multi-Level Disk . . . 88
6.4 $\mathcal{APT}$: Approximating $\mathcal{PT}$ . . . 89
6.4.1 Experiment 5: $\mathcal{LIX}$ vs. $\mathcal{APT}$ . . . 90
6.4.2 Experiment 6: Sensitivity of $\mathcal{APT}$ . . . 92
6.5 Related Work . . . 94
6.6 Conclusions . . . 95

7 Disseminating Updates **97**
7.1 Introduction . . . 97
7.2 Models of Consistency . . . 99
7.3 Consistency Techniques . . . 101
7.3.1 Invalidation . . . 102
7.3.2 Auto-prefetch . . . 103
7.3.3 Propagation . . . 103
7.4 Experiments and Results . . . 104
7.4.1 Parameter Settings . . . 104
7.4.2 Major Cycle Update . . . 106
7.4.3 Continuous Update . . . 109
7.4.4 Refining the Results . . . 111
7.5 Related Work . . . 113
7.6 Conclusions . . . 114

8 Balancing Push and Pull **116**
8.1 Introduction 116
8.2 Integrating a Backchannel 118
8.2.1 Algorithms 119
8.2.2 Assumptions 121
8.3 Experiments and Results 121
8.3.1 Parameter Settings 122
8.3.2 Experiment 1: Basic Tradeoffs 122
8.3.3 Experiment 2 - Reducing Backchannel Usage 128
8.3.4 Experiment 3 - Restricting the Use of Push 130
8.3.5 Summary of Results 133
8.4 Related Work 134
8.5 Conclusions 135

9 Designing a Broadcast Disks Prototype **137**
9.1 Introduction 137
9.2 Key Differences with the Simulator 138
9.3 Design Description 139
9.3.1 Server Design 140
9.3.2 Client Design 144
9.4 Experimental Results 146
9.4.1 Parameter Settings 147
9.4.2 Simulation Validation 149
9.4.3 Cache Management Overhead Experiments 151
9.4.4 Server Cache Experiments 158
9.5 Conclusions 160

10 Conclusions **162**
10.1 Summary of Results 162
10.2 Future Work 165

A Proof of Optimality of $\mathcal{PIX}$ **167**

Bibliography **171**

List of Tables

2.1 Some Examples of Bandwidth Asymmetry 15

3.1 Expected Delay For Various Access Probabilities 29

4.1 Client Parameter Description 55
4.2 Server Parameter Description 57
4.3 Use of Δ in a 3-disk broadcast program 57
4.4 Update Parameter Description 59
4.5 Backchannel Parameter Description 60

5.1 Basic Parameter Settings 65

6.1 Access and Broadcast Frequencies for Figure 6.1 80
6.2 Prefetching Parameter Settings 82
6.3 Approximation Error vs. Length of Run for $\mathcal{APT}$ 92

7.1 Update Parameter Settings 104

8.1 Backchannel Parameter Settings 121

9.1 Prototype Parameter Description 148

List of Figures

1.1 Pull and Push Data Delivery 7

3.1 A Flat Broadcast Program 27
3.2 Three Example Broadcast Programs 29
3.3 Deriving a Server Broadcast Program 32
3.4 Memory Hierarchy in a Broadcast Setting 36
3.5 Use of Cache to Reduce Client-Server Mismatch 40
3.6 Cost-based Caching Order 43
3.7 Tag-team Caching 44

4.1 Controlling the Logical to Physical Page Map 58
4.2 Read vs. Update distribution 59
4.3 Server Backchannel Model 61

5.1 Client Performance, No Cache, No Noise 65
5.2 Noise Sensitivity, 2-level Disk, No Cache 67
5.3 Noise Sensitivity, 3-level Disk, No Cache 67
5.4 *Offset* Sensitivity, 2-level Disk 68
5.5 *Offset* Sensitivity, 3-level Disk 68
5.6 Noise Sensitivity, $\mathcal{P}$ Policy, Large Cache 69
5.7 Noise Sensitivity, $\mathcal{PIX}$ Policy, Large Cache 69
5.8 Page Replacement in $\mathcal{PIX}$ 70
5.9 $\mathcal{P}$ vs. $\mathcal{PIX}$ Performance, Varying Noise, Large Cache 72
5.10 Access Locations for $\mathcal{P}$ vs. $\mathcal{PIX}$ 73
5.11 Page Replacement in $\mathcal{LIX}$ 73
5.12 $\mathcal{LIX}$ Performance, Δ Sensitivity 74
5.13 Access Locations for LRU, $\mathcal{L}$ and $\mathcal{LIX}$ 75
5.14 $\mathcal{LIX}$ Performance, Noise Sensitivity 76

6.1 `pt` vs. Time 80

6.2 $\mathcal{PT}$ imitates Tag-team case 83
6.3 Effect of Scatter 84
6.4 $\mathcal{PT}$ Performance, Uniform Access, 3-level Disk, Small Cache 85
6.5 $\mathcal{PT}$ Performance, Uniform Access, 3-level Disk, Medium Cache 85
6.6 Cache Residency for $\mathcal{PT}$ 86
6.7 $\mathcal{PT}$ Performance, Skewed Access, 3-level Disk, Small Cache, Varying *Noise* . . . 87
6.8 $\mathcal{PT}$ Performance, Skewed Access, 3-level Disk, Large Cache, Varying *Noise* . . 87
6.9 $\mathcal{PT}$ Performance, Skewed Access, 3-level Disk, Small Cache, Varying Δ 88
6.10 $\mathcal{PT}$ Performance, Skewed Access, 3-level Disk, Large Cache, Varying Δ 88
6.11 Page Replacement in $\mathcal{APT}$ 90
6.12 $\mathcal{APT}$ Performance, Small Cache 91
6.13 $\mathcal{APT}$ Performance, Large Cache 91
6.14 Learning Sample Sensitivity, Small Cache 93
6.15 Learning Sample Sensitivity, Large Cache 93
6.16 $\mathcal{APT}$ Response for different #Regions 94

7.1 Once a Cycle Updates, Update Matches Read Access 106
7.2 Once a Cycle Updates, Update does not Match Read Access 107
7.3 Client Performance for Continuous Update 110
7.4 Refined Update Algorithms, *Noise* Sensitivity 112

8.1 Steady State Client Performance 123
8.2 Client Cache Warm Up Time, $\mathcal{IPP}$ *PullBW* = 50% 126
8.3 Noise Sensitivity, $\mathcal{IPP}$ *PullBW* = 50% 126
8.4 Influence of Threshold on Response Time 128
8.5 Restricting Push Contents, *ThinkTimeRatio* = 25 131
8.6 Server Load Sensitivity for Restricted Push 132

9.1 Prototype Server Block Diagram 141
9.2 Broadcast Page Structure 142
9.3 Prototype Client Block Diagram 145
9.4 Prototype Experiment, Noise Sensitivity, No Cache 149
9.5 No Cache Comparison of Simulator and Prototype 149
9.6 Caching Study, Prototype vs. Simulation 151
9.7 Client Response Time (in seconds) 151
9.8 Response Time (*Broadcast Units*) vs. Broadcast Bandwidth 153
9.9 Response Time (seconds) vs. Broadcast Bandwidth 153
9.10 Caching Overhead, Slow Network 154
9.11 Caching Overhead, Fast Network 154

9.12 Page Drop% vs. System Load 158
9.13 Effect of Page Drop 158
9.14 Bandwidth vs. Cache Size 159
9.15 Server Cache Residency 159

Chapter 1

Introduction

Over the last few years, there have been dramatic changes in the computing and communication landscape. The phenomenal growth of the Internet, the World Wide Web and the availability of high-bandwidth links into the home and on the road via satellite and cable television networks have revolutionized the way data is delivered and distributed between computers. Due to these technological improvements, the cost of transferring data, both monetarily and in terms of bandwidth, is falling rapidly and this is giving rise to newer and bolder applications. For example, applications like Stock Market Monitors, Traffic Information Systems, live audio and video telecast and commercial offerings such as Pointcast [Poi97] and AirMedia [Air97] are becoming more and more commonplace. These applications usually require efficient dissemination of information in real-time and often, have upward of 100,000 clients simultaneously accessing the data. This ability to support such massively large scale applications and the availability of such applications to the common user would be inconceivable a few years back.

However, these advances in processing power and communication along with the explosion in the scale of the internetwork is beginning to test the limits of many assumptions traditionally made when designing distributed computing systems. The combination of these factors is bringing forth a number of new and interesting challenges forcing us to re-evaluate how to design and implement large scale distributed systems. In this thesis, these new developments and the issues that they raise are first highlighted. Then, a novel *dissemination-based* approach for data distribution and delivery is proposed and studied in the context of *client-server* systems, a very common model for distributed computing. Dissemination is a fundamentally different "push-based" approach to data delivery, in contrast to the traditional "pull-based" approach. The underlying theme of all the studies in this thesis is to develop techniques to improve client performance in a manner that is scalable and that adapts to the new tradeoffs in this emerging computing landscape.

1.1 Background and Motivation

Designing a distributed computing system consisting of a large number of cooperating computers requires efficient coordination among all the machines. The client-server model is a simple yet powerful framework for doing cooperative processing. It was originally developed to structure communication among processes and now is the basis of all large scale systems including distributed databases, operating systems, CAD and management information systems. In this section (and the thesis), we will study the client-server model in the framework of database systems.

1.1.1 Client-Server Model

A network of computers can be viewed as a distributed information service resource. In the network, some computers are designated as the service providers, or the ***servers***. Depending on the environment, they are responsible for providing one or more services such as communication, mail, file access, database access, and so on. The computers or applications which receive these services are the ***clients***. When the client requires a service, it make a request to the server. The server receives the request, processes it and either performs the service or, returns an error code. For example, typical client requests to a file server include open, close, read, write or delete a file. The clients and the servers are usually connected to each other using a local or a wide area network.

The role of a database server is to provide fast and efficient access to a large data repository. In the event of a failure, the server is also responsible for efficient recovery to a consistent database state. The client is the machine on which the database applications run. Typically, the role of the client includes providing an interface to pose queries, forwarding them to the server for execution and displaying the output from the server.

In the early days of computing, the database management system (DBMS) was *centralized* with all system components residing at a single site. This meant that the crux of all the processing and computation was the responsibility to a single mainframe server. The users interacted with the system via “dumb” terminals which had very rudimentary display capabilities and processing power.

However, due to falling prices in the early 1980’s, personal computers and workstations became commonplace providing significant compute cycles at very low cost. Simultaneously, network technology improved tremendously making communication very reliable. These developments had two important consequences. One, the increasing reliability in communication and storage created a trend towards *distribution* of computer systems and services over multiple sites connected via a communication network. This trend also extended to DBMSs giving rise to ***Distributed Database*** systems. Two, the proliferation of powerful workstations with significant amounts of local storage created a natural thrust to move some of the computation out of the mainframe onto the workstations. Consequently, the "dumb" user terminals in the mainframe model were upgraded to workstation like machines which would do a significant amount of processing in addition to collecting and displaying query results. *Client-Server DBMSs* are based on such a model.

Like any client-server model, in a client-server DBMS, clients send requests for data to the server and the server responds appropriately. Depending on the granularity of the interaction between the clients and the server, there are two architectural choices for client-server DBMSs [Fra96]. The traditional approach favored by *Relational Database Systems* [Cod70, SKS97] is a *query-shipping* architecture. In query-shipping systems, clients send the query (in either plain text or in a compiled representation) to the server; the server processes the query and delivers the results back to the client. In contrast, the more recent approach taken by *Object-Oriented DBMSs* [ZM90] is a *data-shipping* architecture. In this case, the bulk of the query processing occurs at the client and the server is responsible for delivering the data items required for successful execution of the query. Thus, the client handles significantly more DBMS functionality here than in the query-shipping approach. Data-shipping systems can further be classified as *page-servers* or *object-servers* based on the unit of data transfer. In the former case, clients request physical units of data (e.g., pages or disk segment) while in the latter case, the interaction uses logical data units (e.g., tuples or objects). A detailed description of each of the above architectures and their tradeoffs is presented in [Fra93, Fra96].

In this thesis, we study the client-server environment in the framework of a data-shipping page-server architecture. In the next section, we highlight some of the issues that need to be addressed when designing and implementing such a system.

1.1.2 Issues in Client-Server DBMS Design

The issue of "Terminals or Computers" for a client machine has been often studied in various contexts [DR92, RD91, FJK96, AA95]. In the framework listed above, this question loosely translates to a query-shipping or a data-shipping architecture. As the various studies and benchmarks have shown, the appropriate choice is strongly driven by the nature of the application. In this section, we highlight the pros and cons of using a data-shipping model with a workstation "computer" for a client machine.

The use of powerful workstations with significant memory and compute power for client machines have obvious performance and scalability advantages. While the memory, storage and the processing power on a server is typically much higher than any single machine on the network, the total storage and processing capability of all the workstations combined can easily outstrip the server. Thus, goal of the client-server approach is to improve the performance and scalability by offloading work from the central server onto the much larger resources in this global client "pool".

The performance benefits arise due to lower network and server loads. The memory available at the client allows it to make a copy of the data at the server and store it locally or, "cache" data (described next). This has two benefits. One, the application is more responsive since the data is closer to it. Secondly, the network traffic is greatly reduced by obviating the need to go to the server at every access. This translates to lower server loads and better responsiveness. A lower server load also creates greater scalability. Since many of the server responsibilities are offloaded to the client, the incremental cost on the system of adding a new client is much lower. Thus, the system can support a larger client population before either the server or the network becomes a bottleneck.

However, the downside of the data-shipping approach is that the client can be considerably more complex. In addition to handling applications and providing an interface to communicate with the server, the client is also responsible for significant database functionality including query processing, transaction management and buffer management. Also, if a page-server architecture is used, the client has to maintain a map of logical objects which are queried to physical pages on the server. Two of these issues, namely, buffer and update management, are directly relevant to the studies in this thesis. In the rest of this section, we will highlight them in the page-server DBMS framework so as to put it in context when they are investigated further in the following chapters.

Client Cache Management

The case for using client caching as the fundamental technique to improve performance and scalability in a client-server DBMS has been made in [Fra96]. Caching is a technique of using the local client memory to store popular server pages in order to improve the responsiveness of an application. As explained above, this improvement in performance is because the data is "closer" to the application and due to lower communication overhead.

While caching is a very simple idea, it requires various client mechanisms to make it effective in practice. Typically, the size of the client cache is much smaller than the size of the database. Thus, the client can only cache a subset of the data. Therefore, the basic problem is to determine which pages should be cached locally at the client to achieve the best performance. Alternatively, once the client cache is full and a new page is accessed, what page should be evicted (the "replacement victim") to make space for the new page? This is commonly referred to as the *Page Replacement Problem*.

There are a number of page replacement algorithms proposed in the literature [Tan92]. For example, First In First Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU), Second Chance algorithm, Clock etc. LRU is a very popular algorithm because it can be easily approximated in practice. In its pure form, LRU maintains a linked list of all the pages in the cache. When a page is accessed, it is moved to the top of the list. LRU always chooses as a victim, the page at the bottom of the list (i.e., the least recently used page).

In the standard approach to cache management, the contents of the cache change only on a *cache miss*, i.e., when the page requested by the application is not cache-resident. In other words, standard cache management is passive; it reacts only to misses. An alternative proactive approach to cache management is called *Prefetching*. In prefetching, the cache manager requests a page *before* it is accessed instead of waiting for a miss. When the page is eventually accessed, there is no delay since it is already cache-resident. Prefetching is, thus, a more opportunistic mode of operation than standard caching. Prefetching has been shown to significantly improve the responsiveness of applications [PZ91, CKV93]. However, since every prefetched page uses up cache space, prefetching creates an inherent space-performance tradeoff and should not be done overzealously.

In this thesis, we revisit the problem of standard caching (and prefetching) in a non-traditional client-server system. We show that some of the assumptions on which standard cache management algorithms (like LRU) are based, no longer hold in this new context. We propose some alternate caching strategies which adequately address these new tradeoffs.

Update Management

Caching can be considered to be one form of *second-class replication* [KS92]. In other words, even though the cached copy is an exact replica of the original at the server, the two copies are not "equal" in importance. At any time, the server can ignore the client copy (if, for example, it is unavailable due to a client crash) without affecting the correctness or the consistency of the database state.

Thus, a client-server DBMS has to address the same problems that a replicated database faces in the presence of updates. The influx of updates creates the possibility for inconsistent data since the updates are posted at the server and the clients access their (possibly out-of-date) local copies. Thus, the responsibility of the server in this environment includes informing the clients about the update as soon as possible. Traditionally, there are two techniques used to achieve this in page-server architectures.

A simple, yet efficient, technique is an *invalidation* message. In response to an update, the server sends out a notification with the identifier of the modified page. On receiving the invalidation, clients typically evict the stale page from their cache. Since invalidations only contain an identifier, the message is space and time efficient. The alternative is a *propagation* message which also sends out the new contents of the page along with the invalidation message. In response, the client usually swaps the old and the new versions of the page in the cache. Invalidations and propagations create an interesting tradeoff. Invalidations are efficient, but they destroy the stability of the cache by forcing the client to evict pages independently of the page replacement algorithm. Propagations, on the other hand, preserve the state of the cache but have significantly higher overhead due to the larger message size. Note that invalidations suffice to maintain correctness whereas propagations are intended to enhance performance.

Both the techniques use *common* resources (the network bandwidth and server compute cycles) in order to serve a *single* client. Since a propagation message has a higher overhead, it has a less favorable cost to benefit ratio. Thus, propagation is superior to invalidation only if the pages being propagated are of significant interest to the client in the long run. Otherwise, they waste bandwidth with little performance benefit for the client.

While invalidations and propagations can prevent access to inconsistent data on a per-page basis, the system also has to ensure correctness semantics over updates to multiple items. In other words, in terms of correctness, the client-server architecture has to provide the same level of transaction support as more traditional database architectures. Correctness in database systems is typically based on the notion of *serializability* [GR93]. Serializability, however, imposes strict constraints and very few applications either want or can even support such stringent requirements. Thus, a lot of effort has been invested in defining weaker forms of correctness [DGMS85, ABGM90, PL91].

In theory, a client-server architecture with unlimited (client) cache sizes has a potential for tremendous scalability in a read-only world. In practice, however, the presence of updates and the, consequent actions to ensure correctness, limit the benefits of adding additional client resources. In fact, how efficiently updates can be supported, critically determines the performance of the client-server system. In this thesis, we investigate these issues arising due to updates in a slightly different context from traditional client-server systems. We also explore a number of alternative correctness criteria which are suited to this environment under study.

1.1.3 Data Delivery Choices

As pointed out earlier, in a client-server system, the server is the repository of the data and the clients are the consumers of the data. Thus, when a client application requires a data item, it has to be delivered from the server to the client. Broadly, there are two ways to achieve this data transfer:

1. *Client-initiated Data Delivery.* In this approach, the client requests a data item from the server on *demand*, i.e., when the client application requires it. In response to the request, the server locates the data item and transfers it to the client. To make this transfer feasible, the clients and the server typically use a mutually agreed upon request/response protocol like the Remote Procedure Call (RPC) [Tan92]. As part of the protocol, the client is allowed to make a predetermined set of "requests" (e.g., read a data item, delete a data item etc.) to which the server "responds" appropriately.

2. *Server-initiated Data Delivery.* In this approach, the responsibility of transferring the data rests with the server. Here, the server delivers data items to the client without any explicit request from it, and *in anticipation of an access in the future*. Thus, unlike in the first approach, the transfer is initiated by the server and not the client. Of course, the server

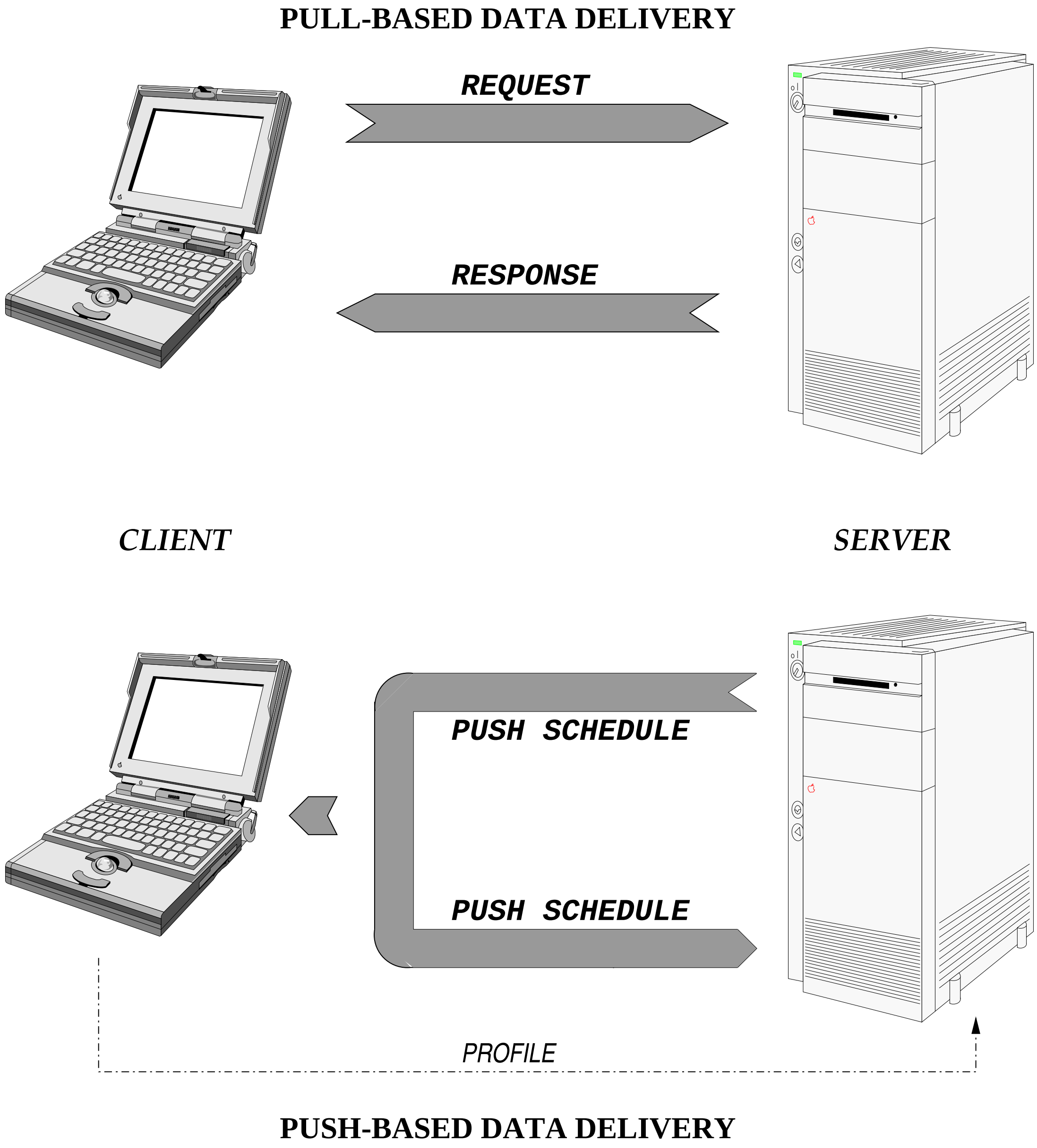


Figure 1.1: Pull and Push Data Delivery

has to have some knowledge of the client data requirements for this method to work well.

The client-initiated delivery is also called *pull-based delivery* and systems built based on it are termed ***Pull-based Systems***. In effect, in this approach, clients "pull" data from the server as needed on demand. The pull-based model is currently the most commonly used mechanism for data delivery. Conversely, the systems using the server-centric approach are called ***Push-based Systems*** since the server "pushes" data out to the clients. These two different models are shown in Figure 1.1.

The push-based approach is analogous to the model used by broadcast television. The television viewing audience is presented entertainment programs daily along with a schedule listing those programs for every channel. Each member of the viewing public, however, does not have direct control over what is shown each night. Nonetheless, the television network chooses the programs based on the ratings which are a direct reflection of what the audience wants to see[1]. A push-based system follows the same model. Based on a characterization of the data items likely to be used by each client (i.e., *profile*), the server generates a push "program" to satisfy their needs. As in television broadcasting, this "program" is a sequence of data items which is preceded by a schedule (i.e., index) which lists the contents and the order of the data sequence. In the simplest scenario, the server just takes a union of all data items requested and transmits them to the client(s).

From the server's viewpoint there are two options for the "programs" that it can generate — *periodic* and *aperiodic* programs. In the periodic case, the server would broadcast the same program repeatedly in a cyclic fashion whereas in the aperiodic case there is no such constraint on the server. Consider, a stock price ticker application for the push model. If the stock server cycled through the prices of all the stocks one at a time, it would constitute a periodic program. Alternatively, if the server pushed out every transaction occurring on the stock market, it would be an aperiodic program due to a lack of repeatability. Also, the characteristics of the data delivery channel determines who can access the push program. If the channel is inherently a *broadcast* medium, like ethernet or wireless networks, the program is visible to all the clients. Alternatively, the channel could be *dedicated* (or *point-to-point*), like a telephone connection and in that case, only a single client has access to the pushed data. Thus, television programming can be considered an example of the *aperiodic broadcast push model*. These concepts will be revisited again in the next chapter.

Since the dawn of distributed computing, the pull-based model has served as the primary mode of data-delivery. However, due to the enormous strides made in computing and network technology over the past few years, the drawbacks of the pull-based model are beginning to emerge. In the next chapter, we will highlight these factors which limit the utility of

[1] Albeit, not instantaneously and in a very global fashion when it periodically re-orders the schedule.

a pull-based approach and motivate why the alternative push-based approach is becoming attractive for many new applications in this emerging environment.

1.2 Contributions of the Thesis

In this thesis, we broadly address the issue of performance and scalability of client-server systems. These issues have, of course, been investigated earlier in the context of databases [Fra93, FC92, DR92, WN90, Sto90] and also, in other areas like operating systems and file systems. However, while the basic problem being addressed is not new, the environment in whose context the problem is being studied is novel. In the recent past, there have been dramatic changes in the nature of computing and communication[2]. In particular, a new class of "asymmetric" environments have now appeared, in which the effective bandwidth from the server to the clients is significantly higher than in the reverse direction[3]. In a client-server system, the server is the data repository and the clients (and their applications) are the consumers of the data. A client-server system is, therefore, only as efficient as the data delivery mechanism from the server to the client. Thus, with the goals of performance and scalability being unchanged, the fundamental problem being addressed in this thesis can be rephrased as:

What is an efficient mechanism to deliver data to clients in asymmetric environments?

The motivation behind this work is the realization that the traditional pull-based approach of data delivery has a potential for severe scalability and performance problems for many emerging applications. In this thesis, we instead take a fundamentally different push-based approach which attempts to avoid the pitfalls of the traditional techniques.

In the following chapters, we investigate a number of new client and server-side mechanisms in pursuit of this goal. This thesis also lays the groundwork for designing client-server systems which use *both* the push and the pull models for data delivery. The rest of the section highlights some of the key contributions of this thesis.

1.2.1 Viability of Push-based Systems

The traditional approach of system design has been pull-based. In this thesis, we motivate why the rise of "asymmetric" networks and the scale of the emerging distributed information systems makes the traditional pull-based client-server model no longer viable for a number of new application domains. We demonstrate that for these domains, a push-based model is

[2]These changes which are relevant to this thesis will be discussed in detail in the next chapter.

[3]The definition of *asymmetry* is broader than just *bandwidth* asymmetry. The next chapter explores this issue further.

not only more efficient in terms of performance but also significantly improves system scalability [AFZ97]. However, neither push or pull is a strictly superior model for all application domains. One of the contributions of this work is to highlight the various tradeoffs of using push versus pull and to delineate the space where pull is a better choice than push and vice versa.

The feasibility of push-based data delivery has, of course, been demonstrated earlier (e.g., Datacycle [BGH+92], Teletext Systems [Won88]). In this thesis, we demonstrate the viability in a more general framework using a detailed simulation-based study and a working prototype. Unlike the previous implementations which typically have used custom components, the prototype uses off-the-shelf hardware and standard software protocols to gauge the usability of current technology for building push-based systems.

1.2.2 Broadcast Disks : A Model for Periodic Broadcast

Another contribution of this thesis, is to present the **Broadcast Disks** model [AAFZ95], a framework for doing *periodic broadcast push*. Recall that the periodic broadcast refers to a push program that is regular, repeats cyclicly and is delivered to all the clients. The Broadcast Disks model is simple yet a very intuitive model for organizing data on the broadcast. It models the broadcast as a number of interleaved "disks", each of different sizes and spinning at different speeds. Using this formulation, the Broadcast Disks model is able to deliver data items to the clients at different rates commensurate with their utility to the client population. In spite of its simplicity, it is extremely flexible and is a generalization of the traditional approach which is the degenerate case of this model.

1.2.3 Cost-based Cache Management

As pointed out in Section 1.1.2, client caching is a very important mechanism for improving performance and scalability in client-server systems. Traditionally, cache management has been based on local probability of access — pages which are accessed the most often at the client, tend to get fixed in its cache.

In this thesis, we demonstrate that probability-based caching fails to exploit the special characteristics of the broadcast environment and thus, performs poorly. Instead, we propose and demonstrate the efficacy of a novel approach to cache management called **cost-based caching** [AAFZ95]. The cost-based model is not only shown to be theoretically optimal but also easily implementable in practice.

The studies also investigate the utility of prefetching in a push-based environment [AFZ96b]. Recall that, unlike in demand caching, prefetching is a more opportunistic use of the cache in which items are pre-loaded in anticipation of an access in the future. Prefetching in this environment, opens new opportunities for utilizing cache space than is possible in demand-driven caching. We demonstrate that when the lessons of cost-based caching are extended to prefetching, it is able to fully exploit the peculiar characteristics of the broadcast medium and thus, provide better performance than demand caching.

1.2.4 Update Management and Cache Consistency

Every client-server environment has to address the update problem and consequently, the issue of maintaining client cache consistency. In [Fra96], this problem is studied in great detail in the context of traditional pull-based client-server systems. In this thesis, we address the same issue in the Broadcast Disks framework. The key contribution of the update study is the discovery that the broadcast environment responds very differently to the presence of updates than the traditional pull-based system [AFZ96a]. Some of these differences are as follows: 1) Unlike in traditional pull-based systems, the broadcast environment was found to be extremely robust in the presence of updates and very simple enhancements provided close to read-only performance. 2) Invalidation-based strategies have been shown to routinely outperform propagation-based strategies in traditional client-server systems [Fra96]. In this environment, due to very different tradeoffs, propagation almost always gives better performance than invalidation. 3) Client-server systems typically use either invalidations or propagations for cache maintenance. This study illustrated that, in this environment, it was possible to have good performance by interleaving both invalidations and propagations simultaneously.[4] Also, performance could be preserved even if propagations were delivered much more infrequently compared to invalidations.

1.3 Organization of the Thesis

The outline of the rest of the thesis is as follows:

• In the next chapter, the concept of push delivery is described in more detail. We introduce the notion of asymmetric communication and motivate why the rise of these asymmetric networks and the increasing scale in causing a gradual shift from the traditional pull model to a push-based model. We also define *data dissemination* which is the basis of the studies in this thesis.

• Chapter 3 introduces the Broadcast Disks model, a framework for doing periodic broadcast. It describes the issues that arise in designing broadcast programs and the algorithm to

[4] This hybrid approach while applicable even in traditional client-server systems, is rarely ever used. An exception is [CFLS91] which proposes an algorithm (subsequently improved in [FC92]) which allows the server to choose dynamically between an invalidation and a propagation. However, the dynamic nature of the algorithm requires client feedback and thus, entails a significantly higher overhead than either of the two native approaches.

generate such a program. Also, the role of client caching in such a framework and its difference from caching in traditional systems is highlighted. The chapter also introduces the notion of Cost-based Caching and makes a case for doing optimistic prefetching in a broadcast environment.

• Chapters 5, 6 and 7 present the key technical contributions of this thesis for the push-only scenario. These chapters contain studies on caching and non-caching client performance, prefetching and influence of updates in this environment.

• In Chapter 8, we augment the Broadcast Disks environment by introducing a backchannel which the clients can use to pull data from the server. The results in this chapter illustrate the various tradeoffs involved in trying to balance push and pull data delivery.

• Chapter 9 describes our experience in building a Broadcast Disks prototype.

• Finally, in Chapter 10 we summarize the scope and contributions of this thesis and discuss various avenues of future work which expand and complement the research presented in this thesis.

Chapter 2

The Case for Push-based Systems

Traditional client-server systems have been based on a pull model of data transfer. In this model, when clients require a data item, they "pull" it from the server by making an explicit request. The pull model is very intuitive and its simplicity makes it an attractive choice. However, over the last few years there have been some very unique and in a sense, revolutionary developments in the computing world. These developments have raised doubts about the suitability of the pull model for a number of evolving applications and have focussed attention on the push-based model of data delivery. Recall that, unlike pull, in the push framework, the server initiates the data transfer and the clients pick up data items as they are transmitted. Thus, the push model obviates the need for clients to make any explicit requests to the server.

In this chapter, we will first highlight these new developments which have precipitated this change in focus. Then, the drawbacks of the pull approach will be listed. Section 2.2 explains how the push approach addresses some of these shortcomings of the pull model and argues why it is a better choice for many existing and emerging applications. Finally, a taxonomy of various models for push delivery will be presented and the notion of *data dissemination*, the basis of the results in this thesis will be introduced.

2.1 Recent Trends (or, Why a Change in Focus?)

2.1.1 Communication Asymmetry

Definition. *In any client-server system, the clients and the serves are connected together by a network. The* upstream *channel refers to the connection from the client to the server. Conversely, the* downstream *channel is connection from the server to the clients. The terms* backchannel *and* uplink *channel will also be used to refer to the* upstream *channel. Similarly, in the rest of the thesis, the* frontchannel *and* downlink *channel will be interchangeably used to mean the* downstream *connection.*

In developing traditional computing systems, designers have made an implicit assumption about the nature of the communication medium. It has always been assumed that the communication capacity (i.e., bandwidth) available to each machine to receive data from the network (i.e., downstream channel) is equal to the bandwidth available to transmit data to the outside world (i.e., upstream channel). This logically followed because machines on the network typically shared the same physical medium ("wire") for both the uplink and downlink communication.

However, recently a number of new network technologies have come to the fore which have the property that they are *asymmetric*. In a client-server framework, bandwidth asymmetry occurs in an environment in which the *effective* bandwidth from the servers to the clients on the downstream channel is significantly higher than the bandwidth available in the reverse direction on the upstream connection. Examples of such environments include satellite and cable television networks.

Asymmetry can also arise for reasons other than bandwidth. The various types of asymmetric communication that are possible are as follows:

- *Network (Bandwidth) Asymmetry* — This is the most obvious form of asymmetry. As explained above, this arises due differences in bandwidth between the upstream and the downstream channels. This bandwidth anomaly occurs because of how the network is set up. In some environments, two physically separate media with different bandwidths are used for the uplink and the downlink connection. For example, satellite networks often require the use of a wired channel (e.g., phone network) for the uplink connection. Alternatively, in many technologies, a single physical channel is used but the bandwidth is disproportionately split among the uplink and the downlink connection (e.g., CATV and phone networks). Table 2.1 lists some asymmetric technologies and the bandwidths they provide in both directions. As can be seen in the table, in all these cases, there is at least an order of magnitude difference between the downstream and the upstream bandwidth.

- *Service Load Asymmetry* — This form of asymmetry arises when a few machines in the network, usually the servers, service significantly more messages than the rest of the nodes in the system. Typically, these messages are requests from clients for some service from the server machine. Since a server has fixed resources and thus, only a finite capacity to handle requests, asymmetry of load can create performance problems due to the inability of the server to respond to requests in a timely fashion. Service load asymmetry can arise for two reasons:

    - ❖ *Client to Server Ratio* — Any environment which has a few servers supporting a much larger client population has a potential for service load asymmetry in the

Network Type	Example Technology	Downstream Bandwidth	Upstream Bandwidth
Satellite	DirecPC [Dir96]	400 Kbps	56.6 Kbps (Via Telephone)
Cable Television	Cable Modems	10-30 Mbps	128 Kbps (Shared)
Telephone	ADSL, VDSL	2-6 Mbps	9.6-640 Kbps
Wireless		1-10 MBps	9.6-19.2 Kbps

Table 2.1: Some Examples of Bandwidth Asymmetry

system. For example, a very popular web server (e.g., the Olympics site or the Wimbledon Tennis Tournament site) faces such an asymmetry.

- ❖ *Updates and New Information* — Asymmetry in load can also arise in environments where new data items are created or existing data items are routinely updated. In this case, there is a potential for asymmetry due to continual polling of the server by clients for the most recent value of the data.

- *Data Volume Asymmetry* — A third way that asymmetry arises is due to the volume of data that is transmitted in each direction. Information retrieval applications typically involve a small request message containing a few query terms or a URL (i.e., the "mouse and key clicks"), and result in the transfer of a much larger object or set of objects in response. For such applications, the downstream bandwidth requirements of each client are much higher than the upstream bandwidth requirements.

From the above list, it should be clear that asymmetry can arise not only due to the properties of the communications network, but can arise even in a "symmetric" environment due to the nature of the data flow in the application.

Disconnection

In many environments, it is not uncommon to have only uni-directional communication from the servers to the clients. Since the clients effectively have no bandwidth to connect to the servers, this situation can be considered to be the *extreme case of communication asymmetry*.

In wireless, satellite and mobile environments, the client machines are typically very low powered portable devices whereas the servers have significantly more computing power and have direct access to a continuous power source. Thus, in such environments, the servers usually have a much greater reach than the clients creating potential for only uni-directional communication.

However, the inability of the clients to reach the server may also be by design. For example, clients in satellite-based pager networks and teletext systems [AW85] are set up to be receive-only with no transmission capabilities due to the infrastructural constraints of setting up a backchannel[1].

In mobile and wireless environments, clients may often have the ability to connect to the server, but due to battery limitations, may choose not to do so. Such a state is called *client disconnection*. During disconnection, clients have the ability to monitor the server transmission but cannot (rather, do not) make any uplink communication[2]. Another battery conservation measure is for the mobile clients to switch to *doze mode* where the disk is spun down and the processing is halted. However, as in disconnection, the machines have the ability to monitor the broadcast. As required, the clients "wake up" to process new data, reverting to duplex communication.

In many applications, there are external non-technological factors which force a one-way communication. For example, in many battlefield applications (described next), soldiers in the field are forbidden from uplinking so as to avoid revealing their physical location to the enemy. In these and all the above scenarios of extreme asymmetry, the server has to anticipate the current requirements of the client populations and deliver data efficiently to the clients without any immediate request (and/or feedback) from them.

2.1.2 Information-Feed Applications

Another recent development is the rise of a number of novel *information-feed* or *information-centered* applications. These applications typically involve the dispersal of new data from a set of *producers* to a (typically) large set of *consumers*. The data in these applications is usually of a public nature; thus, creating a very large client base. Examples of such applications include:

- Advanced Traffic Information Systems (ATIS) — ATIS provides traffic and route planning information to cars specially equipped with computers. As pointed out in [SL96], an ATIS systems may have to serve over 100,000 clients in a large metropolitan city during rush hours.
- Stock Market Monitors [OPSS93].
- News Delivery [YGM95, Poi97, Bac97] and Sports Ticker applications.
- Video-on-demand and other entertainment delivery applications
- Live Audio and Video Telecast — Lately, a number of television news networks are simultaneously broadcasting important events live over the Internet in parallel with the

[1]Many new pager networks are now beginning to provide limited two-way communication.

[2]Some researchers define disconnection as the state when the client cannot send *or receive* any data [BAI93]. Such a situation arises if the disconnection is not *elective* but forced by communication failure.

regular telecast. While the audio and video quality may be poorer than the network broadcast, the real-time guarantees of the transmission are still maintained.

- Microcell Applications — These are futuristic applications where information will be disseminated via the wireless medium over extremely small areas ("microcells"). For example, such applications would be used in store fronts or grocery store aisles to announce the products which are on sale. Customers with specially equipped portable machines (either handheld or in the shopping cart) would be able to view this sale information as they walk by.

- Battlefield Applications — The battlefield of the future is another domain which requires efficient dispersal of information. Soldiers on the field would be expected to carry sophisticated electronic equipment capable of receiving and processing high volumes of data. In such a scenario, the battlefield server(s) would be responsible for efficiently disseminating strategic warfare information. This application is particularly challenging because of the real-time nature of the information, the need for secure transfer and mobility of the clients. Also, in many scenarios, the ability to contact the server may be severely restricted at the clients (soldiers and armored vehicles) for the fear of giving up their location.

These information-based applications have become feasible (and extremely popular) due to the recent availability of high bandwidth links into the home and on the road. The ability of cable and phone networks to carry data and the growing success of mobile computing has made these applications very widespread and also responsive enough to be effective in practice.

2.1.3 Scale

Of all the changes that are affecting the computing landscape, the one with the greatest impact from the viewpoint of distributed information systems is that of *scale*. The dramatic fall in the price of computers along with the proliferation of service providers who provide very cheap access to the Internet, has created a vast global network of interconnected machines which is growing at an astonishing pace. For example, in the year between January 1996 and January 1997, the size of the internet grew by 70% [NW997]. Moreover, the size of the World Wide Web is growing at an even more rapid rate. In April 1995, when NSFNET stopped running the primary backbone of the Internet, the Web accounted for 24% of all the traffic on the backbone.

This growing scale has an impact on nearly every aspect of distributed computing. In a client-server paradigm, it increases the number of clients per server requiring more powerful machines to handle the load. Scale increases the *heterogeneity* of the environment and creates bandwidth bottlenecks. In the next section, we will highlight the issues relevant to this thesis which are impact by this burgeoning scale.

2.2 Why Push?

In this section, we discuss why the push model of data delivery is becoming attractive for a number of application domains. We first highlight the shortcomings for the pull delivery model. The push-based approach not only avoids these drawbacks of the pull model but also has some inherent benefits which make it extremely attractive for many of these applications and this is the focus of Section 2.2.2.

2.2.1 Drawbacks of Pull Model

The pull model of data access is based on a request/response framework. When the application accesses a data item not available in the local cache, clients make a *request* to the server for the item. The server receives the request, retrieves the data item and transmits it to the client (the *response*). Retrieving a data item is only one of many requests that a server may honor. Examples of other requests include deleting, creating and updating a data item. Typically, a protocol similar to RPC (Remote Procedure Call) is used for the client-server communication.

The pull-model is extremely intuitive and has been the basis of most computing systems. However, in spite of its simplicity, the pull model has a basic prerequisite which can significantly restrict its utility in the new computing landscape. For any pull based system to be viable, the clients *require a backchannel to make requests to the server*. This requirement has the following two ramifications:

- *Fewer Application Domains* — As pointed out in the last section, the number of asymmetric environments which provide only uni-direction communication from the server to the clients ("disconnection") is growing rapidly. Also, in many cases clients may have restricted access to the backchannel either due to battery constraints in wireless networks or due to environmental constraints such as in a battlefield. In such domains, the request/response mechanism of the pull model is not a viable option since clients cannot request for new data items.

- *Backchannel Congestion* — The presence of a backchannel also makes it liable to be a bottleneck, severely degrading performance. In traditional client-server systems backchannel congestion is usually not an issue for two reasons — the size of request is typically very small, and the client population can be strictly controlled. However, restricting who can access a server is a significantly more challenging task for applications with huge number of clients like ATIS or on the WWW. Two other factors can further add to congestion. If the channel is shared by all the clients (e.g., ethernet or wireless networks), the potential for bottleneck increases further. The second (and more critical issue) is the inherent asymmetry in many of the new networking technologies (Table 2.1).

Another debilitating consequence of the request/response paradigm is that of *Server Saturation*. Every server machine has an upper limit on the rate at which it can service requests. Let this be the *service rate*. If the rate at which the requests flow in, the *request rate*, is greater than the rate the server can service them (the *service rate*), the server eventually will saturate. In other words, a server is said to be *saturated*, if the following inequality holds,

$$Request\ Rate > Service\ Rate.$$ [3]

Once the server is swamped, it can never catch up with the requests unless either the request rate reduces or the service rate can be improved. Typically, a client-server system is designed by first estimating the maximum request rate and using a server which has a higher service rate. In traditional applications where client-server systems are routinely used, the size of the client population is known *a priori* and consequently, the average request rate. Thus, avoiding saturation is not difficult using brute force solutions.

However, due to the growing scale of the Internet and WWW, many of the emerging applications listed in Section 2.1.2 typically have a huge number of clients. For such applications, predicting the average request rate *a priori* is not trivial. Also, in many of these applications, such as in a Traffic Information System or on the WWW, there is often a wide variation in the system load (e.g., rush hour versus non-rush hour traffic). Thus, when these applications are supported by a pull-based client-server model, there is often inordinate delays in client response time when the current load is significantly higher than the load for which the system was designed. For example, Web and FTP servers routinely stop responding when the service they provide suddenly become very popular (e.g., new software releases or news servers on election nights). The only option available to avoid saturation, is to increase the service rate by adding more (powerful) servers. Unfortunately, hardware-based approaches have their inherent drawbacks. Designing a system for the worst case scenario can be a waste of resources and monetarily expensive if the average load is much lower than the worst case load and/or if it only occurs very infrequently. More importantly, there are upper limits to the server load even for the larger powerful servers.

Thus, the request-response pull model suffers for severe scalability problems and the potential hardware-based solutions are limited in their success. The next section makes a case for using the alternate push-based approach for delivering information and argues why it not only scales better but also can provide better performance.

[3] Often, due to transient effects, this condition may hold for very short periods of time. This does not imply saturation.

2.2.2 Advantages of Push

The arguments made above about the shortcomings of the pull-based model, by itself make a case for the use of push-based systems for many of the emerging applications and environments. In a push framework, the responsibility for data transfer rests with the server. When client applications require a data item, they do not explicitly make a request for it but instead wait for the server to send it out.[4] Thus, the absence of the need for a backchannel automatically rids the push approach of drawbacks of the pull model. Moreover, it opens up a number of application domains which have only uni-directional connection from the server to the clients. The push-based approach, however, also has some inherent advantages which supplement the above arguments to make push the data delivery model of choice for these new applications.

The server-centric approach of the push model makes is attractive for a number of applications where the server alone is aware of the entire state of the system. Examples of such applications include the stock market monitors, traffic systems and other information-feed applications. In these applications, new data is created and updates are posted at the server. Thus, for such applications, the server is better positioned to decide if (and when) to send out data as opposed to the pull-approach where the clients have to continually poll the server for this information.

The prerogative of the data transfer resting with the server in the push-based approach has a three-fold advantage:

- *Efficiency.* Data is transfered only as needed when new data is created or if updates take place minimizing client and server processing which can eventually lead to better performance. The push model is particularly attractive for applications which generate new data since clients cannot be expected to request data items whose existence they are not aware of.

- *Scalability.* Client and server load is also reduced by obviating the need for polling. This directly translates to greater scalability since the server can support more clients before saturation.

- *Lower Bandwidth Utilization.* By avoiding unnecessary data transfers, both the upstream and downstream bandwidth is saved in this approach. The is particularly useful for asymmetric environments.

A data delivery model has an additional scalability advantage if *broadcast* delivery is used for there data transfer where all clients receive the page simultaneously. This form of push

[4] Of course, for any push system to be feasible in practice the server needs to be aware of the client requirements. Also, this mode of delivery is useful only if the delay between when the client wants a data item and when the server transmits it, is not inordinately long. The issues involved in setting up a push-based system will be investigated in greater detail in the next chapter.

delivery is called *dissemination* and it is introduced in the next section.

Thus, the above inherent benefits of the push approach combined with its ability to mitigate the drawbacks of the pull approach make a case for push-based data delivery in the evolving landscape. Of course, studying the tradeoffs of push versus pull and identifying when one is a superior mode of delivery is an important exercise. However, investigating the push model alone and the issues involved in designing a push-only system is interesting on its own right due to the rise of a number of environments which have the may suffer from disconnection (Section 2.1.1). Since environments with this property are only growing in number, understanding how to design and deploy a push-based client-server environment is an immediate challenge which needs to be addressed (in addition to its tradeoffs vis a vis a pull-based approach). In this thesis, we explore both the issues – first, we study a push-only environment and then, investigate the push versus pull tradeoffs.

There are a number of models for doing server push. In the next section, we will introduce one such model, *Data Dissemination*, which is the basis for all the studies in this thesis.

2.3 Data Delivery Models

Push and Pull are not the only ways to categorize the various data delivery options [FZ96]. The data transfer from the server to the clients can be further classified based on if it is:

- Periodic or Aperiodic
- Point-to-point (Unicast) or One-to-many (Broadcast)

Definition. *In a push-based system, the server transmits data to satisfy the needs of the client population. The sequence of the data items transmitted by the server is called the* push program*. If the transmission is over a broadcast channel (described below), the push program is referred to as a* broadcast program*. The term* schedule *will also be used interchangeably to refer a (broadcast or push) program.*

In periodic delivery, the server push program follows a regular, repeating schedule. This periodic delivery could either be in response to a repeating client request pattern (i.e., polling) or initiated by the server itself. Aperiodic delivery has no such repeatability constraints.

The third classification is based on the communication model used by the server to deliver data to the clients. If the server uses a one-to-many model of communication, every data item has multiple recipients. In the literature, there are two models of one-to-many communication — *broadcast* and *multicast* [Tan92]. A broadcast message is delivered to all the clients in the network while a multicast message is intended for only a subset of the client population. Thus, multicast is a special case of the broadcast model. In this thesis, we shall concern ourselves only with the broadcast model and use it as a synonym for the one-to-many model. In contrast, in a point-to-point or an unicast model, each data item has exactly one recipient[5].

[FZ96] presents a taxonomy of eight delivery mechanisms based on the above categorization. In this thesis, only three of these models are directly relevant — *Aperiodic Point-to-Point Pull*, *Aperiodic Broadcast Pull* and the *Periodic Broadcast Push*.

The default delivery option used in traditional client-server systems is the *Aperiodic Point-to-Point Pull* model. Since clients make a request only in response to a miss, the request pattern is aperiodic. Many of the new communication technologies are designed exclusively for broadcast (e.g., satellite, CATV and wireless networks) and thus, adapting the request/response framework to a broadcast setting also allows for the use of the *Aperiodic Broadcast Pull* model. The *Periodic Broadcast Push* is an example of a framework for push called *Data Dissemination*.

2.3.1 Data Dissemination

In this thesis, we define ***Data Dissemination*** as push-based data delivery using the *One-to-many* model.[6] Data dissemination is an ideal framework for information-feed applications (Section 2.1.2) since in these applications data has to be "disseminated" from the few sources to a large client population.

There are two styles of dissemination — *periodic* and *aperiodic* dissemination. In periodic dissemination, the server repeats a broadcast "program" in a cyclic fashion. A example of periodic dissemination is a stock market monitor which cycles through all the stock values. Television broadcast, on the other hand, is an example of *aperiodic dissemination* since there does not exist any repeatability in the schedule.[7]

As a technique for data delivery, dissemination is extremely scalable. Once a broadcast program is set up, any number of clients can join the system without impacting the performance of other clients. However, while in theory, the broadcast nature of the medium can lead to unlimited scalability, it also suffers from some inherent drawbacks. Since every data item transmitted is delivered to all the clients, it can waste network and client resources unless a significant portion of the client population access the data. Also, as the number of clients with non-overlapping interests increase, the data which needs to be disseminated can grow in

[5] In communication literature, point-to-point and unicast refer to slightly different concepts. Point-to-point describes the nature of the physical connection (e.g., a phone line) whereas unicast refers to the nature of the communication message. Thus, it is possible to have an unicast message on a non point-to-point broadcast medium; the message is visible to all clients even though it is intended for only one client. In this thesis, we are concerned with the nature of the message and not the physical characteristics of the connection. Thus, we shall use the the two terms interchangeably to refer to a message directed to exactly one client.

[6] The one-to-many communication model can be simulated by sending individual point-to-point messages to all the clients. Some researchers have argued for a broader definition of dissemination which encompasses information-feed applications which use such a point-to-point model [Fra97]. While simulating the one-to-many model using multiple one-to-one messages achieves the same outcome, the system tradeoffs and cost metrics for such an approach is significantly different than the native one-to-many model.

[7] This is true for most part. However, there are some programs which follow a periodic schedule. For example, the nightly news programs are usually broadcast at the same time each night.

volume. Thus, dissemination is ideally suited for applications which have a strong commonality of interest among the client population. Fortunately, the information-feed applications like stock market monitors and traffic systems (and broadcast television) have this characteristic. Dissemination-based applications can only be expected to gain in popularity as many of the emerging network technologies (CATV, satellite and wireless networks) are designed primarily for broadcast.

In this thesis, the focus is on periodic broadcast push. In the following chapters, we introduce *Broadcast Disks*, a framework for doing periodic dissemination and study the client-server system performance in this context.

2.3.2 Requirements for Dissemination

As explained above, dissemination has obvious scalability and performance advantages for its target applications. However, these benefits come at a price. In particular, a dissemination-based client-server system has the burden of two additional requirements which traditional client-server models do not have to address. In this subsection, we will highlight these two critical requirements. In fact, these requirements are not limited to dissemination *per se* but are valid for push-based systems in general.

User Profiles and Feedback

Definition. *An application* profile *is the data requirements of the application expressed in a suitable form. In the simplest case, a profile may be stated either as an explicit list identifying the data items of interest or as a query whose output is a superset of the data required for successful execution of the application. More advanced profiles impose rankings or complex relationship between the various items of interest.*

In push-based systems, the burden of data transfer is on the server. To be able to deliver the data efficiently, the server needs to know the requirements of the client population. Otherwise, bandwidth may be unnecessarily consumed for pages which have very little demand. In addition to the list of data items which have to be disseminated, the server needs additional information to be able to also prioritize these items based on their popularity (among the clients). This classification is important because critical events (e.g., a stock market crash) require timely dissemination to be useful to the clients.

There are potentially two ways for the server to gain this information. In many applications, there is enough domain knowledge to indicate the size of the data and their relative importance. For example, a traffic system which serves a metropolitan city knows *a priori* the roads whose data it needs to carry. Also, an accident on the main highway is clearly more important than an accident on a side road. However, in many applications this domain knowledge may either not be available or may not reflect the true needs of the client population.

An alternative approach to supplement the domain knowledge is for each user to supply profiles of its access requirements to the server. The server would then collect these profiles and integrate them in some fashion to generate a global profile. Use of profiles requires both client-side and server-side mechanisms. On the client-side, the problem is generating these profiles accurately and on the server-side, the problem is to find a function which aggregates the various profiles fairly.

There are a number of ways by which user profiles can be accrued. A client could monitor its requests over time and use it to build a model. In many applications, it may be trivial to note user interest. For example, in a stock ticker application, users can provide a list of stocks which interest them and the server can use all the lists to generate a global interest model. Alternatively, if the clients have an access to a backchannel to make requests, the server can potentially sample the requests to determine the client interest. Profiles can be expressed in various levels of granularity. They can range from explicit list of items or a predicate to an abstract description of interest (e.g., weather or sport scores). The responsibility of the server is to convert these profiles into a formalism that can be used to query the database. A profile is, in fact, one form of continuous query [TGNO92]. As new data is added to the database (or, old items deleted), the server may need to re-evaluate what appears on the broadcast program. This is exactly the outcome of executing a query continually over an evolving data set.

Accruing profile information is a hard problem and merits further study and is beyond the scope of this thesis. Some initial work on techniques to gauge user interest in a push-based system in available in [Vis94].

Push Program Generation

The problem of deriving a "program" to disseminate data to the clients is another issue which is exclusive to a push-based environment and does not need to be addressed in a traditional client-server system.

There are two steps involved in the broadcast generation process. The first step is the aggregation of various client profiles (expressed in a suitable form) into a single *global* profile. The key issue here is the question of *fairness*. Should all the clients be treated on an equal footing or should some clients receive disproportionately higher priority? This decision is also influenced by the various performance goals of the system. For example, the first option of treating all clients equally will very likely produce the least variance in the average performance of the clients. However, the second option will very likely provide some clients with much better performance at the expense of the other clients. Thus, the performance constraints of the application – best average case performance, best worst case performance etc. strongly influences the fairness criteria that should be used. Also, in some cases many conflicting criteria may have to be supported simultaneously making the aggregation problem only more complex.

The second step is the generation of the actual program from this global profile. Theoretically, this can be considered to be a constraint optimization problem. For example, an instance of this problem can be formulated as follows: given a global profile, it is possible to generate a push schedule which provides an average response time better than some threshold? It should be apparent that the same global profile can produce a number of programs depending on the performance criteria that is being satisfied.

Chapter 3

Broadcast Disks

3.1 Introduction

In a traditional client-server computing environment, each client is responsible for fetching all the data required by its applications from the server. This is typically done using a request/response protocol such as RPC. When the application accesses a data item not available locally, the client sends a request to the server. The server processes the request and responds by delivering the data to the client. Thus, in a traditional pull-based environment, the role of the server (from the standpoint of data delivery) is to monitor to client requests and respond to them efficiently.

On the other hand, in a push-based information system, the only source of new data for the clients is the *push program* generated by the server[1]. Thus, the role of the server in a push-based environment is to construct a program which meet the needs of the client population. In the simplest scenario, given an indication of the data items that are desired by each client listening to the broadcast, the server would simply take the union of the requests and broadcast the resulting set of data items cyclicly.[2] Figure 3.1 depicts a cyclic broadcast program for a set of five pages in a wireless environment.

When an application running on a client needs a data item, it first attempts to retrieve that item from the local memory or disk. If the desired item is not found, then the client monitors the broadcast and waits for the item to arrive. In the simplest scenario, assuming the server generates a *periodic* broadcast program, the expected wait for an item on the broadcast is the same for all items (namely, half a broadcast period) regardless of their relative importance to the clients. This "flat" approach has been adopted in earlier work on broadcast-based database

[1]This assumes client do not have a backchannel to connect to the server.

[2]A program is not constrained to be cyclic. As described in the last chapter it could be periodic or aperiodic, broadcast or a unicast program. Some more options will be listed in Section 3.1.1.

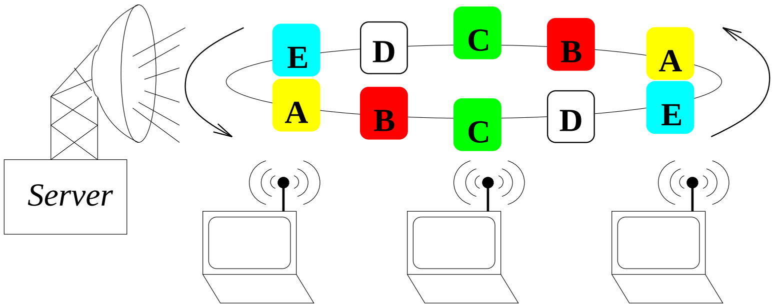


Figure 3.1: A Flat Broadcast Program

systems such as Datacycle[BGH+92] and [IB94].

In real-life, however, data is rarely accessed uniformly. Typically, database accesses tend to be skewed with a small portion of the data (called the *hot spot*) getting a disproportionate number of hits. In a push-based system, the broadcast channel is the source of new data for the client. Thus, rather than being "flat", the broadcast should be structured to mirror the request pattern of the clients. In other words, data items of greater interest (i.e., in the hot spots) should be broadcast more often than items for which there is very little demand. This is the insight and the motivation for the *Broadcast Disks* approach to organizing the broadcast.

Broadcast Disks is a paradigm to structure a *periodic* broadcast program. Recall that, a broadcast is periodic if the server repeats the program over and over again in a cyclic fashion as in Figure 3.1. The length of the single cycle is called the *period* of the broadcast. In the Broadcast Disks paradigm, each data item can have a different broadcast frequency (per period) based on its request profile. A flat program, such as in Figure 3.1, can be visualized as a rotating "disk" on the broadcast channel from which the clients access data[3]. The Broadcast Disks model allows a number of such "disks" each with its own size (i.e., number of data items) and its own speed of rotation (frequency of broadcast)[4]. Note that the flat program is the degenerate case of the Broadcast Disks model - single disk, one rotation per period. In the next section, we highlight some of the desirable properties of a broadcast program and present the details of the Broadcast Disks model. The key challenge is to multiplex the items from various "disks" onto a single channel in a manner that preserves these desirable properties and satisfies the constraints imposed by the broadcast frequency.

3.1.1 Broadcast Program Options

When generating a broadcast program, the server has a number of options available to it. In Section 2.3, we introduced four models of push data delivery – unicast vs. broadcast and periodic vs. aperiodic. In this subsection, we will list the various options available for *periodic*

[3] A "disk" will be formally defined in Section 3.2.2.

[4] Hence the name *Broadcast Disks* — a collection of "disks" on the broadcast channel. Using the same analogy, the client network card or the antenna can be visualized as the disk head which "reads" the data.

broadcast programs, which is the focus of this thesis.

The different styles of periodic broadcast are as follows:

- *Static* or *Dynamic*. A *static* program is a program where the sequence of pages in a cycle remains constant for every cycle. A consequence of this restriction is that each cycle always has the same *period*. A dynamic program has no such constraint.
- *Regular* or *Irregular*. A program is defined to be *regular* if the inter-arrival time between two consecutive occurrences of a page is always the same. If not, it is irregular.

The Broadcast Disks model described next is an example of a *regular periodic broadcast*. In this thesis, we have studied the performance of only the *static* model. However, the structure of a Broadcast Disk can change dynamically and in Section 3.2.3, we list the various options for doing so.

3.2 Structuring the Broadcast Disks

3.2.1 Properties of Periodic Broadcast Programs

The generation of broadcast programs can be theoretically addressed as a *bandwidth allocation* problem; given all of the client access probabilities, the server determines the optimal percentage of the broadcast bandwidth that should be allocated to each item. In [AW85], it has been shown that in the absence of a cache, the optimal bandwidth for each item is proportional to the *square root of its probability of access*. The broadcast program can then be generated randomly according to those bandwidth allocations, such that the *average* inter-arrival time between two instances of the same item matches the needs of the client population. However, such a random broadcast will not be optimal in terms of minimizing expected delay due to the variance in the inter-arrival times.

A simple example demonstrating these points is shown in Figure 3.2. The figure shows three different broadcast programs for a data set containing three pages. Program (a) is a flat broadcast, while (b) and (c) both broadcast page `A` twice as often as pages `B` and `C`. Program (b) is a *skewed* (or, irregular) broadcast, in which subsequent broadcasts of page `A` are clustered together. In contrast, program (c) is *regular*; there is no variance in the inter-arrival time for each page. The performance characteristics of program (c) are the same as if page `A` was stored on a single-page "disk" that is spinning twice as fast as the second two-page "disk" containing pages `B` and `C`. Thus, we refer to program (c) as a *Multi-disk* broadcast. The *Multi-disk* program is representative of a Broadcast Disks schedule.

Table 3.1 shows the expected delay for page requests given various client access probability distributions, for the three different broadcast programs. The expected delay is calculated by multiplying the probability of access with the expected delay for each page and summing the

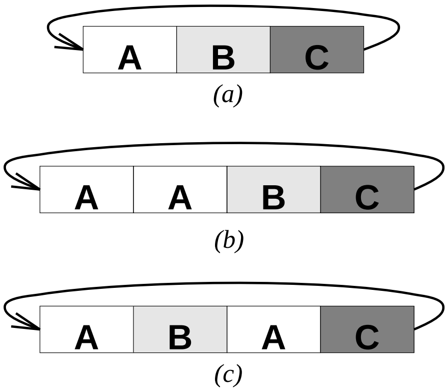


Figure 3.2: Three Example Broadcast Programs

Access Probability			Expected Delay (in broadcast units)		
A	**B**	**C**	**Flat** (a)	**Skewed** (b)	**Multi-disk** (c)
0.333	0.333	0.333	1.50	1.75	1.67
0.50	0.25	0.25	1.50	1.63	1.50
0.75	0.125	0.125	1.50	1.44	1.25
0.90	0.05	0.05	1.50	1.33	1.10
1.0	0.0	0.0	1.50	1.25	1.00

Table 3.1: Expected Delay For Various Access Probabilities

results. For example, consider the delay for the Multi-disk program for the first row. For simplicity assume that requests arrive just when pages are broadcast. The wait for page A is either 0 or 1 page, or, the average wait is 0.5 pages. Similarly the average wait for B and C is 1.5 pages since the actual wait varies from 0 to 3 pages. Since all the three pages have equal probability of access, the total wait is 0.333 * (0.5+1.5+1.5) = 1.17 pages. In reality, the requests may arrive at any time within the broadcast of a single page (and not just at the boundary) and thus, the average addition delay is half a page. This gives the total delay as 1.67 pages. There are three major points that are demonstrated by this table.

The first point is that for uniform page access probabilities (1/3 each), a flat disk has the best expected performance. This fact demonstrates a fundamental constraint of the Broadcast Disk paradigm, namely that due to the total available bandwidth being fixed, *increasing the broadcast rate of one item must necessarily decrease the broadcast rate of one or more other items*.

The second point, however, is that as the access probabilities become increasingly skewed, the non-flat programs perform increasingly better. This follows from the optimal bandwidth formula given in [AW85] — the higher the access probability, the more bandwidth (proportional to the square-root of the probability) a page should be given. However, while a flat program may be unsuited for a skewed access, giving excess bandwidth to a subset of pages also hurts performance. This can be seen in the second row of the table where Page A gets 50% of the accesses. The square-root formula for optimal bandwidth dictates that Page A should get $\sqrt{0.5}/(\sqrt{0.5} + \sqrt{0.25} + \sqrt{0.25}) \approx 41\%$ of the bandwidth. Thus, the flat program gives too little bandwidth (33%) and the Multi-disk program gives too much bandwidth (50%) and therefore, both provide the same non-optimal performance.

The third point demonstrated by Table 3.1 is that the Multi-disk program always performs better than the skewed program. This behavior is the result of the so-called Bus Stop Paradox. If the inter-arrival rate (i.e., broadcast rate) of a page is fixed, then the expected delay for a request arriving at a random time is one-half of the gap between successive broadcasts of the page. In contrast, if there is variance in the inter-arrival rate, then the gaps between broadcasts will be of different lengths. In this case, the probability of a request arriving during a large gap is greater than the probability of the request arriving during a short gap. Thus the expected delay increases as the variance in inter-arrival rate increases.

In addition to performance benefits, a Multi-disk broadcast has several other advantages over a irregular (skewed) broadcast program. First, the randomness in arrivals can reduce the effectiveness of some prefetching techniques that require knowledge of exactly when a particular item will next be broadcast [ZFAA94, AFZ96b]. Second, the randomness of broadcast disallows the use of "sleeping" to reduce power consumption (as in [IVB94a]). Finally, there is no notion of "period" for such a broadcast. Periodicity may be important for providing correct semantics for updates (e.g., as was done in Datacycle [HGLW94, BGH$^+$92]) and for introducing changes to the structure of the broadcast program. For these reasons, we argue that a broadcast program should have the following features:

- The inter-arrival times of subsequent copies of a data item should be fixed.
- There should be a well defined unit of broadcast after which the broadcast repeats (i.e., it should be periodic). As pointed out above, this may be useful for update semantics.
- Subject to the above two constraints, as much of the available broadcast bandwidth should be used as possible.

3.2.2 Broadcast Program Generation

The Broadcast Disks model allows the server to broadcast different items with differing frequency. Such a model emphasizes the most popular items and de-emphasizes the less popular ones. We assume that the server has knowledge of the client requirements and uses that information to derive a global probability model for the data. Also recall that being a page-server architecture, the server broadcasts data items in fixed-sized independent units called "pages".

In the remainder of this section, we present an algorithm to generate a multi-disk broadcast program representative of the Broadcast Disks paradigm. This algorithm generates the program with the desired features listed in the previous section and allows substantial flexibility in fitting the relative broadcast frequencies of data items to the access probabilities of a client population. The algorithm has the following steps:

1. *Order the pages in the database from hottest (most popular) to coldest.*

2. *Partition the list of pages into multiple ranges*, where each range contains pages with similar access probabilities. These ranges are referred to as *disks*. Let the disks be numbered D_1 to $D_{\mathtt{N}}$ where N is the number of disks.

3. *Choose the relative frequency of broadcast f_i for each of disk D_i.* The only restriction on the relative frequencies is that they must be integers. Thus, if $f_1 = 3$ and $f_2 = 2$, pages in Disk D_1 are broadcast three times for every two times pages in Disk D_2 appear in the cycle.

4. *Split each disk into a number of smaller units*. These units are called *chunks* (C_{ij} refers to the j^{th} chunk in disk D_i). First, generate the Least Common Multiple (LCM) of the relative frequencies. This is T_{max}, the maximum number of chunks. Then, split each disk D_i into $T_i = T_{max}/f_i$ chunks. In the previous example, T_1 would be 2, while T_2 would be 3.

5. *Create the broadcast program* by interleaving the chunks of each disk in the following manner:

```
for i := 1 to T_max
    for j := 1 to N
        Broadcast chunk C_{j,(i mod T_j)}
    endfor
endfor
```

Definition. *A* Major Cycle *of the broadcast is defined as the length of a single cycle of the broadcast or, th period after which the broadcast repeats. A* Minor Cycle *is the smallest portion of the program which contains one* chunk *of each of the disks.*

Figure 3.3 shows an example of broadcast program generation for a database of eleven pages. The requirement is a 3-disk program, in which pages in Disk D_1 are to be broadcast twice as frequently as pages in Disk D_2, and four times as frequently as pages in Disk D_3. Therefore, $f_1 = 4$, $f_2 = 2$, and $f_3 = 1$. These disks are split into chunks according to step 4 of the algorithm. Thus, T_{max} is 4 and so $T_1 = 1, T_2 = 2$, and $T_3 = 4$. Note that the chunks of different disks can be of differing sizes. The resulting broadcast consists of four *minor cycles*. The number of minor cycles in T_{max}, the LCM of the relative frequencies. The resulting broadcast has a period of 16 pages. This broadcast produces a three-level memory hierarchy

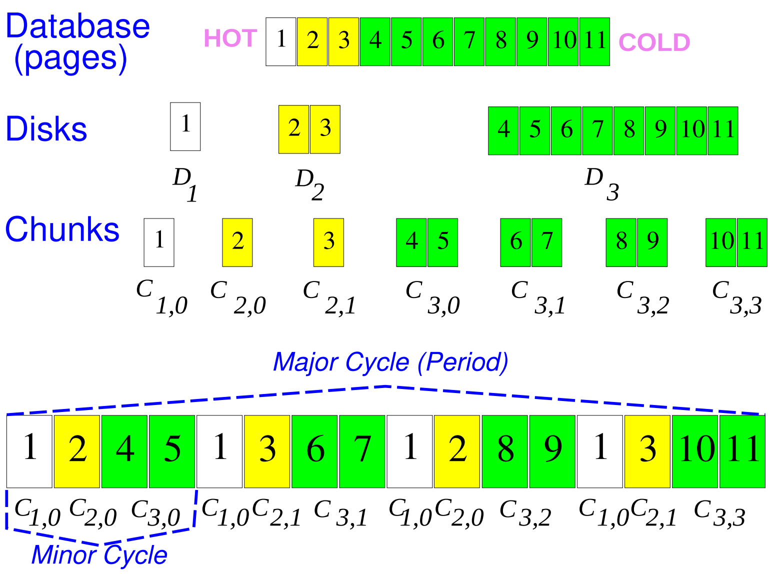


Figure 3.3: Deriving a Server Broadcast Program

in which disk D_1 is the smallest and fastest level and disk D_3 is the largest and slowest level. Thus, the multi-level broadcast corresponds to the traditional notion of a memory hierarchy and this analogy will be explored further in Section 3.2.4.

The algorithm produces a periodic broadcast program with fixed inter-arrival times per page. Some broadcast slots may be unused however, if it is not possible to evenly divide a disk into the required number of chunks (i.e., in Step 4 of the algorithm). Of course, such extra slots need not be wasted, they can be used to broadcast additional information such as indexes, updates, or invalidations; or even for extra broadcasts of extremely important pages. Furthermore, it is anticipated that the number of disks will be small (on the order of 2 to 5) and the number of pages to be broadcast will be substantially larger, so that unused slots (if any) will be only a small fraction of the total number of slots; also, the relative frequencies can be adjusted slightly to reduce the number of unused slots, if necessary.

The disk model, while being fairly simple, allows for the creation of broadcast programs that can be fine-tuned to support a particular access probability distribution. There are three inter-related parameters that can be adjusted to vary the shape of the broadcast. First, the number of disks (`N`) determines the number of different frequencies with which pages will be broadcast. Then, for each disk D_i, the number of pages per disk, and its relative frequency of broadcast (f_i) determine the size of the broadcast, and hence the arrival rate (in real, rather than relative time) for pages on each disk. For example, adding a page to a fast disk can significantly increase the delay for pages on the slower disks. Intuitively, we expect that fast disks will be configured to have many fewer pages than the slower disks, although our model does not enforce this constraint.

Recall that the only constraint on the relative broadcast frequencies of the disks is that they be expressed as positive integers. The advantage of using integer-valued frequencies is that it allows us to generate a regular broadcast with fixed inter-arrival times for pages. As pointed out earlier, regularity may be important to maintain update semantics and to help the client caching heuristics. Since the broadcast frequency determines the bandwidth allocated, all pages on a disk get the same amount of bandwidth. As derived in [AW85], in the optimal program, the bandwidth allocated for any page should be proportional to its access frequency. In other words, the program generated by the above algorithm may not always be *optimal*. In fact, in [JW95], it has been shown that developing an optimal broadcast program for even very simple cases (e.g., each client is interested in only two pages) is NP-complete. Thus, deriving a broadcast program which guarantees the optimal inter-arrival gap (and, consequently the optimal bandwidth) may not be possible except for simple cases. Consequently, this creates an inherent regularity-bandwidth tradeoff for broadcast programs. By constraining frequencies to be integers, we made a deliberate choice of favoring regularity of the broadcast at the expense of less than ideal bandwidth for the various pages. Recently, there have been alternative approaches proposed for broadcast program generation ([VH96, ST97, Chi94, HV97]) which play the tradeoff differently by trying to provide as close to optimal bandwidth as possible at the expense of a loss of the broadcast structure.

While the broadcast program constraints the frequencies to be integers, there are no restrictions on what they may be. Thus, it is possible to have arbitrarily fine distinctions in broadcasts such as a disk that rotates 141 times for every 98 times a slower disk rotates. However, this ratio results in a broadcast that has a very long period (i.e., nearly 14,000 rotations of the fast disk). Furthermore, this requires that the slower disk be of a size that can be split into 141 fairly equal chunks. In addition, it is unlikely that such fine tuning will produce any significant performance benefit (i.e., compared to a 3 to 2 ratio). Therefore, in practice, relative frequencies should be chosen with care and when possible, approximated to simpler ratios.

While the algorithm specified above generates broadcast programs with the properties that we desire, it does not help in the selection of the various parameter values that shape the broadcast. The automatic determination of these parameters for a given access probability distribution is a very interesting optimization problem and is beyond the scope of this thesis. However, prior research on designing broadcast programs for teletext systems [AW85, Won88] and applying solutions to the pin-wheel scheduling problem in real-time systems [BB97] provide guidelines on choosing proper values.

3.2.3 Dynamic Broadcast Programs

In this thesis, we study the impact of *static* Broadcast Disks programs. However, the model is not constrained to have static programs and they change dynamically at every cycle. This may be driven by changes in user profiles or addition (or deletion) of new data to (from) the database or due to updates. In the simplest scenario, at the end of every cycle, the server regenerates the inputs to the Broadcast Disks program based on the currently available data and the usage requirements. Then, using the algorithm given in the last section, a new program can be generated.

There are four possible changes that can be considered in the Broadcast Disks framework:

1. *Item Placement* - Data items may be moved among the levels of a given Broadcast Disks hierarchy (i.e., a set of disks of varying sizes and speeds). Such movement may have an impact on the metrics used by the client-side caching and prefetching policies.

2. *Disk Structure* - The shape of the hierarchy itself may be changed; for example, the ratios of the speeds of the disks may be changed, slots may be added to or removed from a given disk, an entire disk may be added or removed, etc.

3. *Disk Contents* - The set of data items being broadcast changes, i.e., some pages being broadcast are no longer broadcast and/or new items are added to the program.

4. *Data Values* - Updates are made to data items that are already part of the broadcast program. The broadcast remains *static* – only the values of data items in each slot changes.

The first three changes are applicable to both a read-only and an update environment whereas the last change is applicable only in an update scenario.

For *Item Placement* and *Disk Structure* changes, the relative frequencies and/or the order of appearance of the data items already being broadcast are changed. The values of the data items change only if they are updated. In the absence of updates, the impact of these first two types of changes, therefore, is primarily on performance. The metrics used by the caching and prefetching algorithms include information about the latency of data items (described in Section 3.3). Thus, changes due to *Item Placement* and/or *Disk Structure* may cause the relative values of data items to be incorrect, resulting in sub-optimal usage of the client storage resources. Performance problems that might arise from such changes can be mitigated, to some extent, by broadcasting advance notice of the upcoming changes. This information would allow clients to appropriately adjust their caching behavior.

With the third change type, changes to *Disk Contents*, items that used to appear on the broadcast may be removed, while new items may appear. While this type of change does have an impact on the set of data items that appear on the broadcast, it can also be considered to be an extreme case of *Item Placement* change; items are moved to or from a disk having infinite latency. Viewed in this way then, given advance warning of the impending arrival or disappearance of a data item, clients may choose to cache an item before it is dropped from the broadcast or make room for an item that will appear.

The fourth type of change, *Data Value* updates, is qualitatively different from the others, as it raises the issue of *data consistency* (Chapter 7). In Broadcast Disks or any other data-dissemination environment, clients access data from their local caches, while updates to data values are collected at a server site. In order to keep the clients' caches consistent with the updated data values, the client-cached copies of modified items must be invalidated or updated. This is the model we use for the studies in Chapter 7 and the impact of *Data Value* changes are investigated there.

3.2.4 Memory Hierarchy

The storage resources in traditional computing systems have been organized based on the principle of *memory hierarchy* [HP90]. A memory hierarchy consists of different types and sizes of memory. The first level of the hierarchy, which is the closest to the processing unit, consists of the memory module which is the fastest (in terms of access latency) and is usually also the smallest in size. By convention, it is also the highest level. The subsequent lower levels have strictly decreasing speeds of access latencies and typically, increasing sizes. Thus, the hierarchy can be viewed as a pyramid-like structure with lower levels growing in size and decreasing in responsiveness. When the applications need data, the request is satisfied by the highest level closest to CPU which has the data.

In a client-server environment, this hierarchy translates to the client cache, client disk (if any), server cache, server disk followed by any tertiary storage devices. In a push-based system, the broadcast forms an intermediate level of this hierarchy between the client and the server. Thus, when the application accesses a page not locally available (in either the client cache or disk), the broadcast is first used to satisfy the request. If the page is not on the broadcast, subject to the availability of a backchannel, the clients can then make an explicit request to the server.

The multi-level Broadcast Disks paradigm adds an interesting twist to the hierarchy. If the server uses a multi-disk broadcast, this intermediate broadcast level is no longer homogeneous but is a sub-hierarchy in itself. Since the pages on the fastest disk have the lowest latency (on the average), it forms the first level of this sub-hierarchy. The slowest disk is the last level of the sub-hierarchy with the other disks making up the intermediate levels. This multi-level view of the storage is shown in Figure 3.4. In spite of the apparent similarities to the traditional memory hierarchy, this hierarchy in the broadcast setting has the following key differences:

- *Access latency is software tunable.* In a traditional memory hierarchy, the cost of data access is fixed for each level of the hierarchy and is driven by the hardware used for storage at each level. In the Broadcast Disks setting, by appropriately choosing the number of (broadcast) disks, the size of each disk and their frequency of rotation, it is possible to fine-tune the cost of access from each level of the disk. In fact, since the parameters which control the structure of the multi-level broadcast program can be chosen at will,

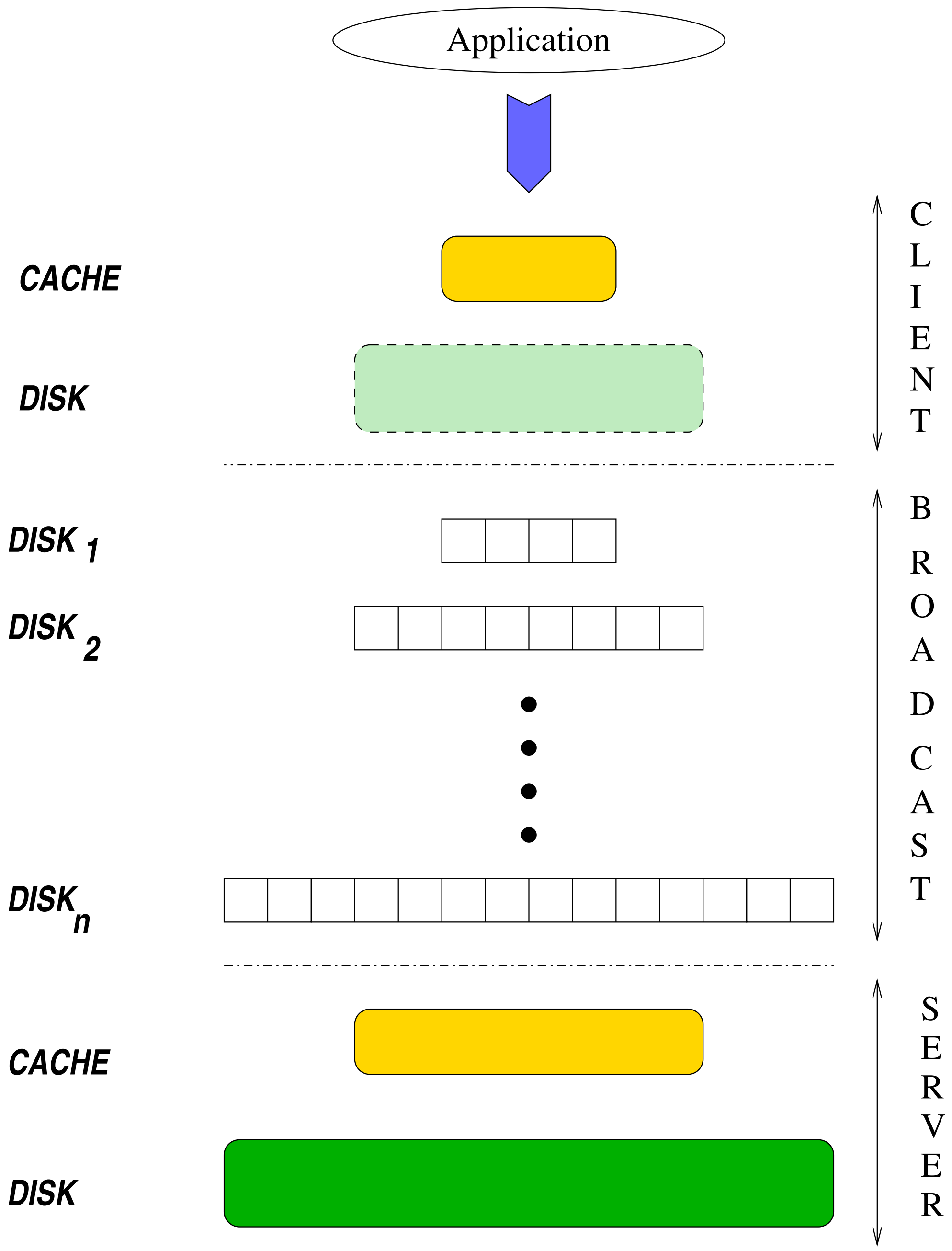


Figure 3.4: Memory Hierarchy in a Broadcast Setting

it is possible to create *an arbitrarily fine-grained memory hierarchy between the client and the server in the broadcast scenario.* Of course, as pointed out earlier, excessive fine-tuning may hurt rather than improve performance in practice.

- *Hierarchy is not strict.* In a traditional hierarchy, both the *average and instantaneous* cost of access from a higher level is always lesser than that from a lower level. In other words, there is a strict order of increasing access cost going down the hierarchy.[5] In the broadcast setting, this strict order is maintained only for the *average* cost of access. The instantaneous cost of access of any page from the broadcast may vary anywhere from zero to the period of the broadcast. Also, if the clients have a backchannel to connect to the server, in many cases, it may be cheaper to fetch the data from the server than wait for it to come by on the broadcast channel.

Viewing the broadcast channel as a component of the multi-level memory hierarchy raises issues akin to those faced by traditional systems with similar complex hierarchies like tertiary and magnetic storage systems [Sar95, AH93, AS92]. However, the differences listed above along with the periodicity of the broadcast cycle create subtle variations to these problems and over the next few chapters these issues will be highlighted and addressed.

3.3 Client Cache Management

In this section, we describe the issues that arise in managing the client cache resources in a dissemination-based environment. Caching is a well known technique used in traditional client-server systems to improve performance and scalability. By storing data close to the application in the client cache, the system improves the response time by minimizing the number of requests to the (farther) central data repository, typically the server in a client-server system.

Broadly, there are two forms of caching studied in the literature:

- *Demand-driven Caching*. This is the most common form of cache management. As the name suggests, in this case data items are brought into the cache on demand, i.e., when they are accessed the first time (resulting in a cache miss).

- *Prefetching*. Prefetching is a more opportunistic use of the cache resources. Here, data items are brought into the cache *in anticipation of an access in the future*. The goal of prefetching is to pre-cache the data item in order to have no delay when the access eventually occurs. However, prefetching can waste cache space if the prefetched page is not accessed soon thereafter.

[5] In many new environments, this guarantee may no longer hold. Due to increasing network speeds, in many client-server systems, it may be cheaper to fetch data from the server cache than from the local client disk [FCL93]. However, such systems are exceptions rather than the rule.

The primary responsibility of any caching strategy is to find a page to replace (called the *replacement victim*) to make cache space for the new page. Since replacement is a very common task, the process of victim selection is very critical to the client performance. It should not only be very space and time efficient but should also be doable in a manner that minimizes the overall cost of data access. Informally, the data item with the lowest *utility of caching* (at that moment) should be chosen as the victim.

The dissemination nature of the broadcast environment moves the burden of data transfer to the server and introduces some key differences which change the tradeoffs typically associated with caching and prefetching in traditional environments. In what follows, we highlight these differences and argue for a fundamentally new approach to managing the client cache.

3.3.1 Key Differences with Pull-based Systems

In a traditional pull-based environment, clients use a request/response protocol (like RPC) to fetch data from the server. When the client faults on a page not in its local storage, it makes an explicit request to the server for the data. In a dissemination-only environment, the clients do not (or, cannot) make an explicit request for a data item due to the absence of a backchannel from the clients to the server.[6] Instead, they listen on the broadcast channel and pick up the data item as it is broadcast. Consequently, from a client's viewpoint, there are two key differences between a traditional pull-based and a dissemination-only environment:

1. *Broadcast channel is serial*. Unlike in traditional systems, clients have *no random access* to the data on the broadcast channel because it is a serial line. Thus, the server program dictates the order in which the data is sent out and the clients are constrained to receive the data in exactly the same order.

2. *Data access cost is not uniform*. In a traditional client-server system, the cost of a cache miss at the client is the round trip time of the request/response protocol. Ignoring the variance in network loads and the effects of caching and disk accesses at the server, the miss latency can be assumed to be the same for all pages. In other words, all pages not in the client cache are *equidistant* from the client. In a dissemination-based system, this equidistance assumption for misses is no longer valid. Due to the multi-disk framework of the Broadcast Disks paradigm, some pages are broadcast more often per cycle than others. Consequently, on the average, the latency of a miss for a page on a faster spinning disk is much lower than that of a slower spinning disk. Also, the instantaneous cost of access of each page is not constant — it continually varies with time depending on the current position in the broadcast cycle. Thus, in a broadcast environment, *pages not in the client cache are not all equidistant from the client*.

[6] The issues that arise in dissemination-based environments which provide a backchannel is studied in Chapter 8.

These two differences introduce new tradeoffs for demand-driven caching and prefetching in dissemination-based systems. Consequently, the cache has to play a fundamentally different role here than in traditional systems. The next section highlights the new role of client caching in a broadcast environment and describes a novel approach called *Cost-based Caching* which incorporates these differences into the caching policy.

3.3.2 Role of Caching in Broadcast Environments

The size of a client cache is typically significantly smaller than the size of the database. Thus, the role of any caching algorithm is to choose an appropriate cache size subset of the database which gives the best responsiveness to the client applications. In traditional, pull-based systems, this is typically achieved by probability-based caching, i.e., by storing the pages with the highest probability of access (the *hottest* pages). As pointed out earlier, due to the *equidistance* property, all cache misses in a pull-based environment have the same performance penalty. Thus, the strategy which minimizes the number of cache misses (i.e., improves the *hit rate*), provides the best performance. Probability-based caching achieves exactly that goal. However, this approach fails to extend to a broadcast environment. The shared nature of the broadcast channel and the differences from traditional systems highlighted in the last section, have a very important implication — namely, *the role of client caching is fundamentally different in dissemination-based systems*.

In a broadcast environment, the server generates a broadcast program taking into account the access needs of the entire client population. The shared nature of the broadcast disk, while in principle allowing for nearly unlimited scalability, in fact gives rise to a fundamental tradeoff: *tuning the performance of the broadcast is a zero-sum game*; improving the broadcast for any one access probability distribution will hurt the performance of clients with different access distributions.

In the rest of this section, we will describe the role of caching in a dissemination-based environment using an abstract example. Consider, a broadcast server S disseminating $\mathcal{D}$ pages for n clients $C_1 \ldots C_n$. Let,

$\mathcal{A}_i$: Access profile for C_i, $i \in n$.

Recall that a profile is a description of the data requirements of the applications running on the client. The first step in the broadcast generation process it the generation if a global access profile by aggregating the access requirements of all the clients. Let the server's global profile be $\mathcal{A}_s$. If ξ is some aggregation function, then $\mathcal{A}_s$ can be defined as:

$$\mathcal{A}_s = \xi(\mathcal{A}_1 \ldots \mathcal{A}_n).$$

Also, assume that the server has the ability to generate the optimal broadcast program

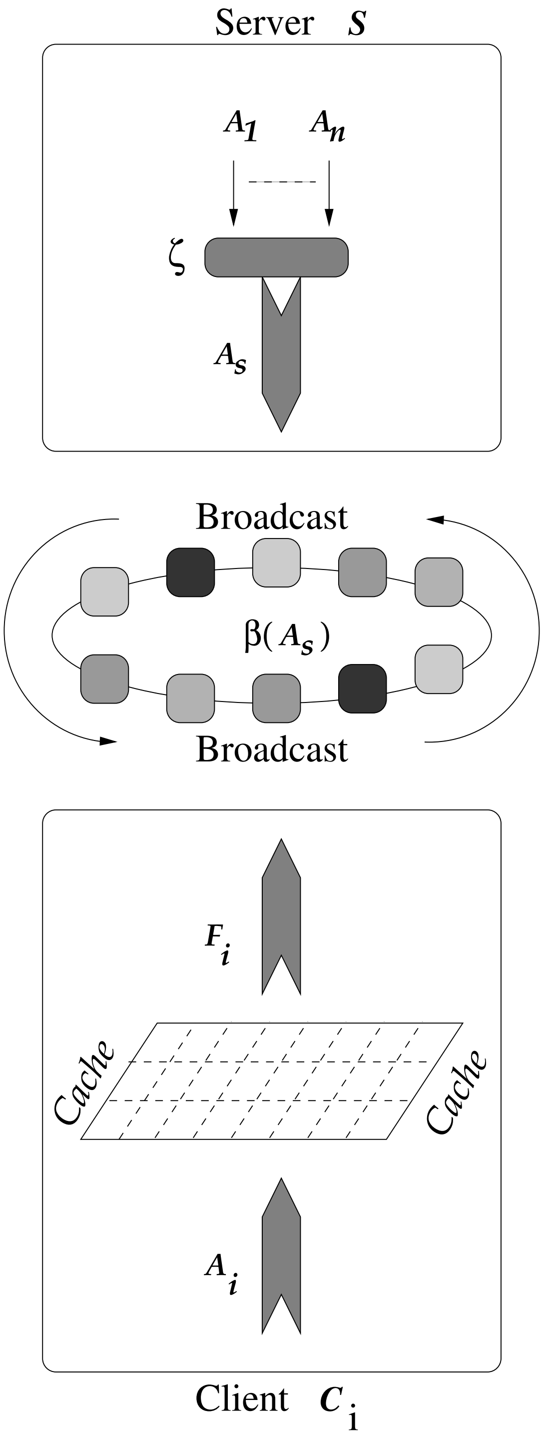


Figure 3.5: Use of Cache to Reduce Client-Server Mismatch

given any access profile.[7]. Let, the optimal broadcast program generated by the server for the access profile $\mathcal{A}_s$ be denoted by $\beta(\mathcal{A}_s)$.

Figure 3.5 shows the process for the server and a single client C_i.

If system has to support only a single client C_1, then the server program $\beta(\mathcal{A}_s)$ will be optimal for the client since $\mathcal{A}_s \equiv \mathcal{A}_1$. In general, however, there are several factors that could cause the server's broadcast to be sub-optimal for a particular client:

- The server may have to average its broadcast over the needs of a large client population.
- The access distribution that the client gives the server may be inaccurate.
- A client's access distribution may change over time.
- The server may give higher priority to the needs of other clients with different access distributions.

Thus, the only way to mitigate the effect of a sub-optimal broadcast is for every client to use its local cache. In fact, the role of the cache in this environment is to improve client responsiveness by *not storing the hottest pages* but instead being a resource to minimize the difference between what the client expects from the broadcast program and what the server actually delivers.

Let for each client C_i,

c_i: Set of pages in the cache of client C_i. Typically, $c_i \subseteq \mathcal{D}$, assuming $|c_i| \leq |\mathcal{D}|$.

If the page a accessed by the client C_i is cache-resident (i.e., $a \in c_i$), the request is satisfied immediately. If not, the client waits until the page is broadcast. Thus, the access pattern visible to the broadcast is a subset of the accesses made — the cache filters the requests for the cache-resident pages. Thus let,

$\mathcal{F}_i$: Filtered access profile for C_i, i.e., $\mathcal{F}_i = \mathcal{A}_i - c_i$.

Consider the case where the client C_i has no cache and accesses the broadcast for every request. So, $c_i = \emptyset$ and $\mathcal{F}_i = \mathcal{A}_i$. If the server pattern $\mathcal{A}_s$ is different from the client C_i's request pattern $\mathcal{A}_i$ (i.e., $\mathcal{A}_s \not\equiv \mathcal{A}_i$)[8], C_i will have a poor performance since the broadcast generated by the server ($\beta(\mathcal{A}_s)$) will be different from the client's ideal broadcast $\beta(\mathcal{A}_i)$. Thus, a cache-less client's performance in such a system is very sensitive to how *close* its access requirements are to the *average* client's access profile, since that determines the global profile $\mathcal{A}_s$.

However, if the client has a cache, it can offset a poorly designed broadcast program (from its viewpoint) provided it can use its cache intelligently. Thus, even if $\mathcal{A}_i \not\equiv \mathcal{A}_s$, client C_i can be

[7] As argued earlier, an optimal program may not always be possible in practice.

[8] The use of equivalence in this discussion is in a very abstract sense. In reality, various statistical techniques can be used the measure the "similarity" or "difference" of two access distributions.

responsive if it can choose a subset of the database c_i to store in the cache, such that $\mathcal{F}_i \equiv \mathcal{A}_s$. In other words, the client should use the cache to filter its accesses in such a fashion that the filtered pattern $\mathcal{F}_i$ is *similar* to what the server uses to generate the broadcast ($\mathcal{A}_s$).

In reality, an equivalence may never be possible. However, the better $\mathcal{F}_i$ can be engineered to be similar to $\mathcal{A}_s$ (by caching appropriately), the better the client C_i's performance. If the server program alters (due to a change in $\mathcal{A}_s$), in order to maintain its performance, the client has to re-adjust its cache-resident set c_i such that the equivalence is satisfied. Therefore, the size of the cache and the variance in the broadcast program determine the degree of the client responsiveness.

The role of cache management in any environment is to improve the client's performance. In a pull-based system, it suffices to just know the local access profile (and store the locally important pages) to achieve this goal. In a broadcast environment, doing so only leads to poor performance. Instead, the role of the cache is to be a mediator between the local client requirements and server schedule (which is a reflection the access needs of the entire population). This is a fundamental difference between the two data-delivery models.

3.3.3 Cost-based Caching

The previous sub-section made the following key observations with respect to caching in a broadcast-environment.

1. Due to the serial nature of the broadcast channel, traditional probability-based caching is unsuitable since it fails to account for the cost of retrieval from the broadcast channel.

2. The cache should be used as a sieve to selectively permit those requests through whose local probability matches the server's global probability of access (which determines how the broadcast is structured).

As pointed out in Section 3.2, the server allocates bandwidth for each page in some proportion to its global likelihood of access. In other words, the frequency of transmission of each page per cycle is a reflection of the server's access probability.

Thus, the client should cache those pages whose local probability of access is significantly different from the server's broadcast frequency and pick those pages from the broadcast whose frequency of transmission is similar to the local access probability. This is shown diagrammatically in Figure 3.6.

Let all the pages of interest for each client be broadly categorized as *hot* (high probability of access) and *cold* (low access probability) pages. Let there also be a similar division at the server. If the client has a cache size smaller than its interest set, Figure 3.6 shows the order in which pages should be stored in the cache (lower number implies higher priority of caching). The client should first cache the pages which are locally important but not popular across all

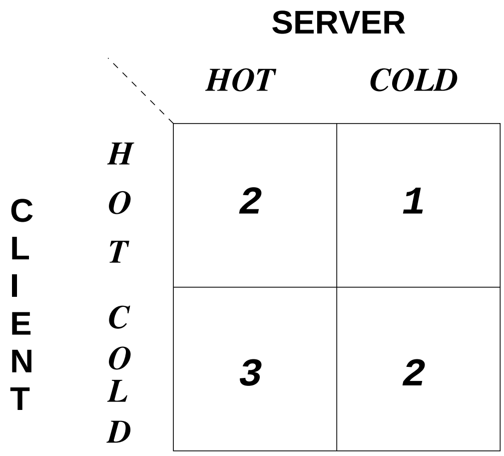


Figure 3.6: Cost-based Caching Order

the clients to warrant a high broadcast frequency, i.e., locally *hot* but globally *cold* pages. Then, the client should has an option of storing either *hot* pages which are also broadcast often or *cold* pages which are not. The choice of one or the other should be done on a case by case basis depending on how important they are to the client (i.e., how hot or cold) and the penalty of missing on those pages (i.e., how often they appear). Similarly, the client should evict pages in the reverse order of caching. Thus, if the client has to choose as a replacement victim among two cached pages which are equally important locally (i.e., *hot* for the client), it should evict the page which is *hot* at the server and thus, has a higher frequency of broadcast. In other words, other factors being equal, the page with a *lower cost of re-acquisition* should be evicted first.

Of course, in reality, the utility of items follow a continuum rather than a binary classification. However, the above argument leads to the need for *Cost-based Caching*. That is, the cost of re-acquiring a page once it is evicted must be accounted for during page replacement decisions. A standard page replacement policy tries to replace the cache-resident page with the lowest probability of access (e.g., this is what LRU tries to approximate) because in a traditional system the cost of re-acquisition is the same for all pages. Instead, in cost-based page replacement, the victim is chosen as the cache resident page which has a low probability of access *and* a low cost of re-acquisition.

In Chapter 5, an optimal cost-based caching algorithm $\mathcal{PIX}$ is presented along with an implementable approximation $\mathcal{LIX}$.

3.3.4 Prefetching

For Broadcast Disks (as in most systems), the goal of prefetching is to improve the response time for the clients. Recall that prefetching is an optimistic cache management strategy where

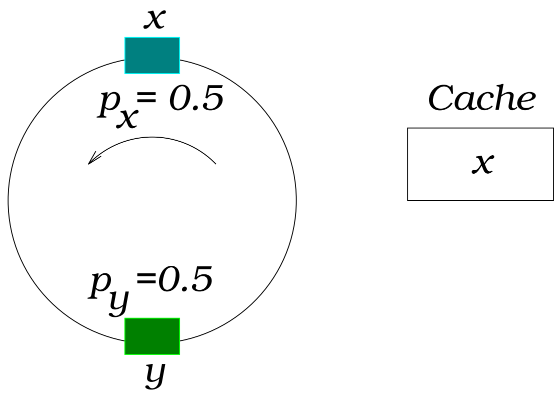


Figure 3.7: Tag-team Caching

pages are brought into the cache *in anticipation of* a future request and not on access as in demand-driven caching. The dissemination-oriented nature of the Broadcast Disks environment makes it particularly conducive to prefetching. In a traditional environment, like databases and file systems, a client has to explicitly request a page for prefetch. In this environment, pages flow past the clients anyway making it worthwhile for it to pre-cache the page if it is important. Also, prefetching in a traditional system places additional load on shared resources (disks, network etc.), which can be potential bottlenecks. In contrast, prefetching in a Broadcast Disks environment only impacts the local client resources. Thus, the risks in prefetching are significantly lower than in a more traditional system. However, the downside of overzealous prefetching still remains — cache slots wasted on pages that are not important or those that are prefetched *too early*.

Response time improvements at the client can be typically achieved in two ways: 1) by fetching as much of the working set into the local cache as possible, i.e., minimizing the cache miss rate, and 2) by reducing the cost of a cache miss. Traditional system designers have focussed considerable energy trying to increase the hit rate (the first approach) since all cache misses have the same penalty. While improving the hit rate can be beneficial in a broadcast environment, greater gains are obtained here by reducing the latency when there is a miss. Cost-based caching, described last section, attempts to address this issue by preferentially caching those pages which have the greatest re-acquisition cost. Extending the lessons of cost-based caching to prefetching, provides additional ways for pages to be brought into the cache which further improve performance by minimize the latency when there is a miss. The potential benefits of this optimistic cache management strategy is illustrated using the following simple example shown in Figure 3.7.

Consider a client that is interested in accessing two pages (x and y) with equal probability (i.e., $p_x = p_y = 0.5$), and has only a single cache slot available. In this example (as shown in the figure), the server places these two pages 180 degrees apart on the broadcast (for simplicity, a single, flat disk is used).

Under a *demand-driven* strategy (standard or cost-based), the client would cache one of the pages, say x, as the result of a request for that page. Subsequent accesses to x can be satisfied locally from the cache, and thus, will incur no delay. However, if the client then accesses page y, it waits for y to come by on the broadcast, and then replaces x with y in the cache. Page y remains cache-resident until x is brought in again on a miss. With this strategy, the expected delay on a cache miss is one half of a disk rotation (because the request can be made at a random time). The expected cost C_i of accessing a page i is given by the formula:

$$C_i = p_i * m_i * d_i,$$

where for page i, p_i is the access probability, m_i is the expected probability of a cache miss and d_i is the expected delay before page i arrives on the broadcast. The expected total cost of access over all pages for this demand-driven strategy is therefore:

$$\sum_{i \in \{x,y\}} C_i = C_x + C_y = 0.5 * 0.5 * 0.5 + 0.5 * 0.5 * 0.5 = 0.25,$$

i.e., one quarter of a disk rotation.

In contrast, consider a simple prefetching strategy that fetches page x into the cache when it arrives on the broadcast and holds it (for half a cycle) until page y is broadcast. At that point, the client drops x and caches page y. When x is broadcast again, y is dropped and x is cached. The expected cost of this strategy, called *Tag-team Caching*, for this example is:

$$\sum_{i \in \{x,y\}} C_i = C_x + C_y = 0.5 * 0.5 * 0.25 + 0.5 * 0.5 * 0.25 = 0.125.$$

Thus, the average Tag-team cost is one eighth of a disk rotation — one *half* the expected cost of the demand-driven strategy. In other words, by prefetching Tag-team is able to double the client performance over demand-driven caching.

It is important to note that the performance gain of tag-team comes not from an improved hit rate (both strategies have a hit rate of 50%), but rather because the cost of a miss using tag-team is half of the cost of a miss under the demand-driven strategy. With the demand-driven strategy, a miss can occur at any point in the broadcast. In contrast, using Tag-team, misses can only occur during half of the broadcast. For example, a miss on a reference to page x can only occur during the part of the broadcast between the arrival (and prefetching) of page y and the subsequent arrival of page x. Tag-team reduces the cost of cache misses by ensuring that a page is cached for the portion of the broadcast when the wait for that page would be the longest. The lesson from Tag-team is that if a cache slot can be allocated to a page for only a portion of the broadcast cycle, the page should be cached for the portion when the page is farthest away from the client (i.e., has the greatest delay).

This simple example demonstrates the significant improvement prefetching can provide over demand-driven caching. However, the performance gain comes at an expense. In demand-driven caching, the broadcast has to be sampled only when there is a miss. Prefetching, on the other hand, requires a client computation on every broadcast page to determine if it is *prefetch-worthy*. This can be computationally very expensive for the client. In Chapter 6 prefetching is explored in greater detail. A "pure" prefetching strategy $\mathcal{PT}$ is introduced along with an implementable approximation $\mathcal{APT}$ and their performance is measured quantitatively.

3.4 Scope of the Thesis

In order to make the studies in this thesis tractable, we had to restrict the broadcast environment in a numbers of ways. In this section, we will list our assumptions. First, the general assumptions which hold throughout the simulation studies in the thesis (Chapters 5–8) will be listed. Then, some of the assumptions which have been selectively relaxed in different chapters will be presented. Each chapter also lists additional conditions specific to the issue at hand which supplement those given here.

The general assumptions include:

- *Periodic Dissemination.* The results presented in this thesis are relevant only in the framework of periodic data broadcast.[9] We assume the communication medium is inherently broadcast and every page transmitted is visible to all the clients.
- *Client profiles are known to the server.* We assume that the clients have informed the server by some means about their access requirements. The server uses this knowledge to generate a broadcast.
- *The client population and their access needs do not change.* This implies that the organization of the broadcast program remains static. Some of the issues that arise when the client requirements change can be addressed once clients have a backchannel as described in Chapter 8. However, a general study of dynamic workloads and consequent, dynamically adjusted broadcast programs is beyond the scope of this thesis.
- *All data items on the broadcast are self-identifying.* This is easily done in practice by adding appropriate information on the headers of each page. Also, as proposed in [IVB94a], the broadcast could be supplemented with an index to speed up the identification and retrieval process.
- *The clients are continually listening to the broadcast.* Monitoring the broadcast consumes power at the client and continually doing so may be restrictive in mobile environments

[9] The periodic constraint is only for the push program. Responses to pull requests (Chapter 8) and update related communication such as invalidations (Chapter 7) have no such restriction.

where battery power is limited [IB94]. However, we allow this restriction since it simplifies the broadcast program design considerably and in many of the other environments where this research is applicable (e.g., CATV and satellite networks), power management is not an issue. If the battery needs to be conserved, the indexing technique referred to earlier [IVB94a], can be used by the clients to "doze" and "wake-up" as and when required.

- *Clients remain active for significant periods of time.* Once a client begins monitoring the broadcast, it does so for a period long enough to fill up its cache and reach steady-state. Thus, in the studies that follow we focus mainly on the steady-state performance of the clients. However, in Chapter 8, this condition is relaxed and the warm-up performance is also studied.

- *Cost of local data access is free.* We assume that it does not cost the client to access data from the local cache. This is a reasonable assumption since memory access latency is negligible compared to the wait for data to arrive on the broadcast for most network technologies. However, if the network speeds rise dramatically, this assumption may have to be re-evaluated.

- *No cache management overhead cost.* Using the same argument above, we assume that the cost of cache management (e.g., choosing a victim, updating cache data structures) is negligible in comparison to the cost of fetching the data from the broadcast. In other words, even though various cache management algorithms have different algorithmic and execution overhead, we ignore those costs in the simulations and compare them as if they entailed no overhead. The validity of this assumption is tested in a realistic setting on the prototype described in Chapter 9.

- *No inter-item dependencies.* In the studies that follow, we use an independent access model for the data. In other words, there are no inter-item dependencies among the different pages in the database. Also, the clients are not transactional and access data items only one-at-a-time.

Unless otherwise noted, these above restrictions hold for all the experiments in this work. In addition to these a few more restrictions are placed, which are relaxed one at a time as the studies progress. These assumptions are listed below:

- *Data is accessed only on-demand*. In other words, the clients perform no prefetching. This assumption was lifted in the studies in Chapters 6– 8.

- *The database is read-only.* This assumptions holds for Chapters 5, 6 and 8. The influence in updates on the data is studied in Chapter 7.

- *Clients have no-backchannel capacity.* This restriction is lifted in the studies in Chapter 8 where clients can provide feedback to the server using an uplink channel.

3.5 Approach of the Thesis

The Broadcast Disks environment is unique in that there is a much stronger synergy between the operation of the client and the server than in traditional systems. Thus, trying to understand the dynamics of this environment and studying the system performance poses a number of new and interesting challenges. Also, the field of dissemination-based systems is only now beginning to emerge as an area of active research and thus, lacks the maturity of the traditional pull-based systems. This thesis is an initial step in laying out the framework for a better understanding of the tradeoffs which govern the efficient functioning and the performance of a dissemination-based environment.

In this thesis, we have taken a *client-centric* approach to studying the environment — given a new client which joins a dissemination-based system (and is presented a server broadcast program), how should it manage its local cache resources, so as to give its applications the best responsiveness? In a real system, the server would be expected to take into account the access pattern of the client population when generating the broadcast. However, akin to the model of broadcast television, it is unlikely that any single client would have control over the actual schedule being generated — the server alone would determine what formulation to use to generate the program from the client input. Therefore, every client cannot expect to receive a program tailored exactly for its access profile. Thus, the focus of this thesis is to develop efficient cache management policies which provide the clients with good performance, given *any* broadcast program. However, issues that arise in designing "good" broadcast program (relative to a given access) are also considered. Besides investigating the client performance for the given program, a running theme in all the experiments is to analyze the sensitivity of the various strategies — how far can the server program deviate from the "ideal" program expected by the client before the cache is insufficient to mitigate the mismatch?

An alternative approach to studying the environment is to take the *server-centric* approach. Given the set of data items and the access requirements of all of the clients, the goal of this approach is design the *optimal* broadcast program which gives the best client performance. As the name suggests, unlike in the *client-centric* approach, the responsibility of improving the system performance rests with the server and not the clients. This approach is attractive in environments with thin clients which have very small caches to offset poorly designed broadcast programs. For example, earlier work [AW85, Won88] on Teletext systems where the television set-top boxes have very little storage have taken this approach. However, the server-centric approach has a number of drawbacks if applied in the Broadcast Disks framework. Firstly, to be effective, the server-side approach requires continual feedback from the clients about changing access needs. The Broadcast Disks framework is designed for inherently asymmetric networks and use of the uplink channel to communicate with the server can be extremely prohibitive (or even impossible in some cases). Secondly, designing an optimal program is a very hard problem and even simple cases have been shown to be NP-complete [JW95]. Also, in reality, the access probabilities of the pages to be broadcast will only be approximate at best. Thus, as a first step, it suffices to generate "good" broadcast programs and instead, invest effort in improving cache management techniques. However, developing efficient heuristics to generate better programs is an interesting optimization problem and an avenue for interesting future work. The next section on related work highlights some of the initial attempts. An extension of the server-centric approach is to use it in combination with the client-centric approach. In other words, both the clients and the server cooperate to give the clients the best performance. However, as the example in Section 3.3.2 demonstrated, the server needs to be aware of the client's caching policy (and what it is likely to keep cache-resident) in order to tailor the broadcast. This may be unrealistic or extremely expensive in many environments especially if it is asymmetric.

The client-centric approach also models real-life applications more faithfully. Consider a traffic information application. In a large metropolitan city, such an application may have to support over 100,000 clients during peak traffic hours [SL94]. If a new client joins the system, it is very unlikely, given the size of the client population that it can significantly influence the server to honor its traffic needs. Instead, this client has to make the best use of the current program broadcast by the traffic server.

Thus, for the reasons listed above, we have taken a client-centric view of exploring the broadcast environment in this thesis. However, use of the server-centric approach and the hybrid mix of both approaches are interesting avenues for further study.

3.6 Related Work in Data Dissemination

The idea of using dissemination as a means to deliver data was first proposed over a decade ago in the early 1980s [GLB85]. However, after a lull of a few years, interest in push-based systems have had a revival with the proliferation of mobile and wireless computing and the widespread use of satellite and cable networks. Over the recent past there have not only been research efforts (described next) but also a number of commercial offerings based on push technology including Pointcast [Poi97], AirMedia [Air97], Marimba [Mar97], and so on. In this section, we will briefly survey the work on data dissemination. The fundamental difference between each of these efforts and the research presented in this thesis is our multi-level approach to data broadcast and its relationship to cache management. In this section, we will broadly describe the various approaches. Following the experiments presented in Chapters 6–8, we will delve into specific details and make comparisons between our approach to solving the issue at hand and the alternative approaches taken by the projects listed below.

One of the earliest applications of data dissemination was for community information services [GLB85, Gif90]. The Boston Community Information System (BCIS) was started at MIT in late 1982 to deliver news and information to clients who had personal computers specially equipped with radio receivers. The project was field tested for about two years in the metropolitan Boston area and had about 200 clients. The system provided both simplex (push-only) and duplex (request/response) communication between the clients and the server(s). The major conclusions of this project was that users valued both the simplex and the duplex form of data delivery and that it demonstrated the validity of the dissemination-based approach for building large scale information systems.

The Datacycle project [HGLW94, BGH$^+$92] at Bellcore was the first push-based approach to building a database system. Datacycle was intended to exploit high bandwidth, optical communication technology and employed custom VLSI data filters for performing associative searches and continuous queries on the broadcast data. Datacycle broadcast data using a flat disk approach and thus, did not address the multi-level disk issues that we have investigated in this thesis. Datacycle provided all the functionality of a traditional DBMS including query processing, transaction management, recovery, and so on. Consistency of the locally cached client data was maintained using a form of optimistic concurrency control and providing some guarantees about the mutual consistency of the data in a single cycle. Datacycle also provided an "upstream network" that allowed clients to communicate with the server.

As television programming began to increase its reach, two new approaches to information delivery – *Teletext* and *Videotext* systems became popular in the mid-80s (particularly in Europe and Asia). Teletext systems used the television infrastructure to provide one-way push-only broadcast while videotext systems allowed for two-way communication. [Won88] is a survey of the theoretical results on designing broadcast programs and caching at clients in the framework of these two systems. [AW85] presents the formula for optimal bandwidth, which states that, in the absence of a cache, the bandwidth for each page which gives the best client performance should be proportional to the *square root* of the access frequency. In [Amm87], the issue of caching is studied in the teletext framework. However, their model of the client is very simplistic and is not applicable in a realistic setting. The request/response framework in videotext systems where the response is broadcast to all the clients (as in the Broadcast Disks model) is studied in [WA85].

The mobility group at Rutgers has studied the problem of data dissemination in a slightly different context [IB94, IVB94a, IVB94b]. Since their focus is the wireless and mobile domain, their attention is directed at conserving battery power on the mobile client machines. Since listening to the broadcast consumes power (albeit, at a much lower rate than transmitting), they concentrated on designing strategies which minimize the client listening time. [IB94] surveys the challenges for data management in mobile computing and argues for the use of dissemination as a viable alternative to deal with the exploding scale. In [IVB94a, IVB94b], a number of indexing strategies have been proposed and studied which allow the client to use the index to determine how long to "doze" before the data of interest arrives on the broadcast. Since listening to the broadcast uses up the battery, use of the index provides a means to "wake up" at appropriate moments to pick up data items. [TX96] extends their indexing scheme to a non-flat framework like the Broadcast Disks model, where different pages are broadcast at different rates per cycle.

[ABGM90] has addressed the issues of caching in dissemination-based information retrieval systems. They introduced the notion of *quasi-copies* at the client which were allowed to deviate from the original at the server in a controlled fashion. Other papers which deal with the problem of updates in push-based systems include [BI94, BGH+92]. The specific differences between these approaches and ours will be explored in Chapter 7.

The problem of generating the broadcast program which provides the best client performance has received a lot of attention very recently. However, most of this work is in the context of weak clients which do not perform any caching. [JW95] shows that developing an optimal broadcast program for even very simple cases (e.g., each client is interested in only two pages) is NP-complete. [AW85, AW87] addressed the problem of designing the broadcast program as a variation of the bandwidth allocation problem. In [AW87], a markov decision process was used to choose the order of pages when generating the program. Recently, numerous heuristics have been proposed along the same lines to generate "good" on-line and off-line broadcast programs [VH96, ST97, Chi94, HV97]. As pointed out earlier, in these approaches the focus is on providing the best response time (by giving as close to optimal bandwidth as possible to pages) at the expense of regularity in the broadcast (in terms of inter-arrival gaps between consecutive occurrences of the same page). On the other hand, in the Broadcast Disks approach, regularity of the broadcast is favored.[10] Since there are pros and cons to either choice, the application requirements should guide the choice of the appropriate schedule.

A number of papers have also considered hybrid approach of delivering data using both a push and a pull delivery mechanism. The results of this study in the Broadcast Disks framework is in Chapter 8. [WD88] addresses this issue in the context of teletext systems and [IV94] offers a number of data delivery choices in the wireless context. As in this thesis, in each of these papers, a broadcast downlink is assumed. A slightly different model has been proposed very recently in [SRB96, SRB97]. Here broadcast delivery is used for the push program but the pull responses go only to the client which made the request (i.e., point-to-point response). In [Bes96a], the notion of *Fault Tolerant Broadcast Disks* was introduced which adds redundancy to the data in order to cope with error in the broadcast.

The World Wide Web also provides immense opportunities for push-based applications.

[10] The benefits of a regular broadcast is highlighted in Section 3.2.1.

However, unlike the papers listed above, the delivery is usually *aperiodic* and due to the absence of support for large scale broadcast, the one-to-many transfer is often simulated using many one-to-one transfers. [Bes96b] demonstrates the utility of source-initiated pushing of documents to servers which are closer to clients than the original source. The case for similar push caching to geographically closer sites has also been made earlier in [GS95]. SIFT [YGM95] is a recently commercialized research prototype developed at Stanford for dissemination of USENET news articles. A host of other commercial products based on dissemination are now available on the internet. These mainly deliver news and information and in some cases specialized data like sport scores and stock quotes. Examples include Pointcast [Poi97], Marimba [Mar97], BackWeb [Bac97], and so on. Each of these applications claim to use "push" technology to deliver the data. However, in reality the client stub of the software periodically *polls* the server to receive updates. Of course, this approach is driven by absence of any mechanism to effectively do push on the global scale of the Internet. Once such a support is available, they can be expected to switch to a native push model. Many dissemination-based applications are also beginning to appear outside the realm of the Internet on wireless and cable networks. Examples include AirMedia [Air97], which delivers information to computers via a specialized wireless interface and Traffic Information Systems [SL94, SL96]. As the additional infrastructure cost (e.g., wireless receivers or cable modems) falls further, such applications can hope to gain greater market share.

Chapter 4

Modeling a Broadcast Disks Environment

The results presented in the following chapters have been based on a detailed and flexible simulation model of the Broadcast Disks environment. The simulator was implemented in C using CSIM [Sch86], a commercial software for process-oriented, discrete-event simulation. The simulation experiments were run using Quahog [Qua97], a batch processing system similar to Condor [LLM88]. Quahog takes a list of experiments and assigns them to be run simultaneously on workstations across the network. By making use of processing and storage resources of multiple idle workstations, Quahog significantly reduces the time to generate results.

The simulator models two key components: a server broadcasting data items and client(s) that continually access them either from the broadcast or their local storage. The unit of data transfer is a uniform fixed-length page and the results are independent of the size of a page. The simulator measures performance in logical time units called *broadcast units*. One broadcast unit is the time required to broadcast a single page. In general, the results obtained from the simulator are valid across many possible broadcast media. The actual response times experienced for a given medium will depend on the amount of real time required to broadcast a page.

In order to speed up the simulation runs, we model only one client for the studies in Chapters 5-7. The ability to capture the multi-client environment by simulating only one client is because of the broadcast nature of the downlink channel. Once the broadcast program is set up, any number of clients can listen in on the broadcast channel and the performance of any single client is independent of the presence of other clients in the system. However, as the number of clients increase, the server broadcast program has to adapt to account for their access needs. Thus, as long as the effect of multiple clients on the server program can be reflected in some fashion, it suffices to simulate a single client.

In the simulator, the client generates requests for *logical* pages. These logical pages are mapped to *physical* pages broadcast by the server. The mapping of *logical* pages to *physical* pages allows the server broadcast to be varied with respect to the client workload. This flexibility to control mapping allows the simulator to model the impact of a large client population on the performance of a single client, without having to model the other clients. For example, having the client access only a subset of the pages models the fact that the server is broadcasting pages for other clients as well. Furthermore, by systematically perturbing the client's page access probabilities with respect to the server's expectation of those probabilities, the simulator is able to vary the degree to which the server broadcast favors the particular client being modeled.

Once the clients have a backchannel to make requests to the server, there is explicit contention for the backchannel and the server resources. In this situation, the simulator has to model multiple clients to account for the impact of this contention. In Chapter 8, which presents the backchannel experiments, the simulator also runs a second client to model the "effect" the other clients in the system.

In this chapter the simulation model is described in detail. First, the client model and the client workloads are discussed. Then, the basic server model is described followed by the extensions needed to account for updates and a backchannel from the clients.

4.1 Client Model

This section highlights the client model. As pointed out earlier, in the Broadcast Disks environment, it suffices to model either one or two clients to study system performance. The next section describes the single client model which is the basis of the results in Chapters 5 – 7. This single client whose performance is studied is called *Measured Client*($\mathcal{MC}$). Then, the model of the second client, the *Virtual Client*($\mathcal{VC}$) is described. Chapter 8 presents the results of the two-client model. The parameter *NumClients* determines the number of clients in the simulation.

4.1.1 Measured Client Execution Model

The parameters that describe the operation of the single client $\mathcal{MC}$ is shown in Table 4.1. $\mathcal{MC}$ runs a continuous loop that randomly requests a page according to a specified distribution. The client has a cache that can hold *CacheSize* pages. If the requested page is not cache-resident, then the client waits for the page to arrive on the broadcast and then brings it into its cache. Client cache management is done similarly to buffer management in a traditional system; if all cache slots are occupied, then a page replacement policy is used to choose a victim for replacement. Once the requested page is cache resident, the client waits *MC_ThinkTime* broadcast

Name	Description
Basic Parameters	
NumClients	Number of clients simulated
$\mathcal{MC}$ **Parameters**	
MC_ThinkTime	Time between client page accesses
AccessRange	# of pages in range accessed by client
CacheSize	Client cache size (in pages)
θ	Zipf distribution parameter
RegionSize	# of pages per region for Zipf distribution
$\mathcal{VC}$ **Parameters**	
ThinkTimeRatio	Ratio of $\mathcal{MC}$ to $\mathcal{VC}$ think times
SteadyStatePerc	% $\mathcal{VC}$ requests filtered through cache

Table 4.1: Client Parameter Description

units of time and then makes the next request. The *MC_ThinkTime* parameter allows the cost of client processing relative to page broadcast time to be adjusted, thus it can be used to model workload processing as well as the relative speeds of the CPU and the broadcast medium.

The client chooses the pages to access from the range 1 to *AccessRange*, which can be a subset of the pages that are broadcast. All pages outside of this range have a zero probability of access at the client. The *AccessRange* parameter is, thus, used to model the fact that the server is generating the broadcast for multiple clients with non-overlapping interests. In this thesis, we assumed two models of client access:

- *Uniform Distribution*: The client accesses all pages in the *AccessRange* uniformly with the same probability.

- *Zipf Distribution* [Knu81]: The Zipf distribution with a parameter θ is frequently used to model non-uniform access [GSE+94]. It produces access patterns that become increasingly skewed as θ increases — the probability of accessing any page numbered i is proportional to $(1/i)^\theta$. Similar to earlier models of skewed access [DDY90], we partitioned the pages into regions of *RegionSize* pages each, such that the probability of accessing any page within a region is uniform; the Zipf distribution is applied to these regions. Regions do not overlap and thus, there are *AccessRange/RegionSize* regions.

4.1.2 Virtual Client Execution Model

This section describes the two-client model. The two-client model is used to simulate an arbitrarily large client population. The first client, the Measured Client ($\mathcal{MC}$), described in the last section is the process whose performance is reported in the simulation experiments. The second client process is the Virtual Client ($\mathcal{VC}$) and it models the *combined effect of all other clients in the system*.

The parameters that describe the operation of $\mathcal{VC}$ are shown in Table 4.1. The Virtual Client $\mathcal{VC}$, like $\mathcal{MC}$, follows a two step "request-think" loop. The parameter *ThinkTimeRatio* is used to vary the relative intensity of $\mathcal{VC}$ request generation compared to $\mathcal{MC}$. The think time of $\mathcal{VC}$ is drawn from an exponential distribution with a mean of *MC_ThinkTime/ThinkTimeRatio*. Thus, the higher the *ThinkTimeRatio*, the more frequent are $\mathcal{VC}$'s requests and the larger a client population $\mathcal{VC}$ represents. Like $\mathcal{MC}$, $\mathcal{VC}$ follows a Zipf distribution to access the database and filters all its requests through a cache of *CacheSize* pages.

When a client begins operation, it brings pages into its cache as it faults on them. This is referred to as the *warm-up* phase of operation. Once the cache has been full for some time, the client is said to be in *steady state*. At any point in time, it is likely that there are some clients just joining the system, some leaving it, and some in steady-state. Since $\mathcal{VC}$ represents the entire client population (minus the single client $\mathcal{MC}$), we assume that some of its requests are from clients in warm-up and some are from clients in steady-state. In warm-up, a client's cache is relatively empty, therefore we assume that every access will be a miss. In steady-state, a client's cache is filled with its most important pages, and thus, we filter all requests through the cache. Thus, the simulation assumes that the clients either have empty caches or full caches. Consequently, in the simulation we generate slightly more misses than in a real system because in reality there would also be clients with partially filled caches.

We use the parameter *SteadyStatePerc* to represent the proportion of clients modeled by $\mathcal{VC}$ that are in steady-state as described above. Before every access made by $\mathcal{VC}$, a coin weighted by the parameter *SteadyStatePerc* is tossed and depending on the outcome the request is either a) filtered through the cache (*steady-state*) or b) directly sent to the server (*warm-up*).

4.2 Server Model

In this section, the model of the Broadcast Disks server is described. First, the basic structure of the server execution model and the parameters which control the broadcast program is discussed. Then, the extensions required to handle updates and a backchannel is highlighted. Two parameters *UpdateOn* and *BackChannelOn* determine if the server model supports updates and honors requests over the backchannel respectively.

4.2.1 Server Execution Model

The parameters that describe the operation of the server are shown in Table 4.2. The server broadcasts pages in the range of 1 to *ServerDBSize*. These pages are interleaved into a broadcast program according to the algorithm described in Section 3.2. This program is broadcast

Parameter	Description
ServerDBSize	Number of distinct pages to be broadcast
NumDisks	Number of disks
DiskSize$_i$	Size of disk i (in pages)
Δ	Broadcast shape parameter
Offset	Offset from default client access
Noise	% workload deviation
UpdateOn	Determines server support for updates
BackChannelOn	Determines server backchannel support

Table 4.2: Server Parameter Description

Δ	f_1	f_2	f_3
0	1	1	1
1	3	2	1
2	5	3	1
3	7	4	1

Table 4.3: Use of Δ in a 3-disk broadcast program

repeatedly by the server. The structure of the broadcast program is described by several parameters. *NumDisks* is the number of levels (i.e., "disks") in the multi-disk program. By convention disks are numbered from 1 (fastest) to N=*NumDisks* (slowest). *DiskSize$_i$*, $i \in$ [1..N], is the number of pages assigned to each disk i. Each page is broadcast on exactly one disk, so the sum of *DiskSize$_i$* over all i is equal to the *ServerDBSize*.

In addition to the size and number of disks, the model must also capture their relative speeds. As described in Section 3.2, the relative speeds of the various disks can be any positive integers. In order to make experimentation tractable, however, we introduce a parameter called Δ, which determines the relative frequencies of the disks in a restricted manner. Using Δ, the frequency of broadcast f_i of each disk i, can be computed relative to f_N, the broadcast frequency of the slowest disk N as follows:

$$f_i/f_N = (N - i)\Delta + 1.$$

Table 4.3 shows the relative spin speeds for a three disk broadcast. When Δ is zero, the broadcast is flat: all disks spin at the same speed. As Δ is increased, the speed differentials among the disks increase. For example, in Table 4.3, when $\Delta = 1$, Disk 1 spins three times as fast as Disk 3, while Disk 2 spins twice as fast as Disk 3. When $\Delta = 3$, the relative speeds are 7, 4, and 1 for respectively. It is important to note that Δ is used in the study only to organize the space of disk configurations that we examine. It is not part of the disk model as described in Section 3.2.

The remaining three parameters, *Offset*, *Scatter* and *Noise*, are used to modify the mapping between the *logical* pages requested by the client and the *physical* pages broadcast by the server. When *Offset* and *Noise* are both set to zero, and the disk is *Unscattered*, then the logical to

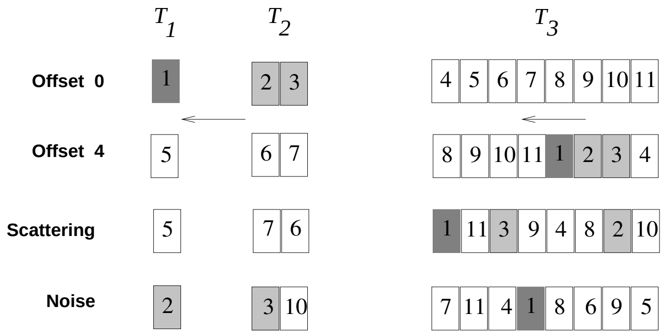


Figure 4.1: Controlling the Logical to Physical Page Map

physical mapping is simply the identity function. In this case, the $DiskSize_1$ hottest pages from the client's perspective (i.e., 1 to $DiskSize_1$) are placed on Disk 1, the next $DiskSize_2$ hottest pages are placed on Disk 2, etc. However, this mapping may be sub-optimal due to client caching. Some client cache management policies tend to fix certain pages in the client's buffer which makes broadcasting them frequently a waste of bandwidth. In such cases, the best broadcast can be obtained by shifting the hottest pages from the fastest disk to the slowest. *Offset* is the number of pages that are shifted in this manner. An offset of K shifts the access pattern by K pages, pushing the K hottest pages to the end of the slowest disk and bringing colder pages to the faster disks.

The second parameter is called *Scatter*, which plays an important role in client prefetching. When *Scatter* is turned on, every page is randomly swapped with another page on its own disk causing a random shuffle of pages within a disk. The goal of *Scatter* is to prevent important pages from being clustered together and distributing them evenly across the entire broadcast.

In contrast to *Offset*, which is used to provide a better broadcast for the client, the parameter *Noise* is used to introduce disagreement between the needs of the client and the broadcast program generated by the server. Disagreement can arise in many ways, including dynamic client access patterns and conflicting access requirements among a population of clients. *Noise* determines the percentage of pages for which there may be a mismatch between the client and the server. *Noise* is introduced as follows: for each page in the mapping, a coin weighted by *Noise* is tossed. If, based on the coin toss, a page i is selected, then a disk d is chosen randomly to be its new destination. To make way for i, an existing page j on d is chosen randomly, and i and j are swapped.[1]

The generation of the server broadcast program works as follows. First, the mapping from logical to physical pages is generated as the identity function. Second, this mapping is shifted by *Offset* pages as described above. Third, if *Scatter* is on, pages in each disk are shuffled among themselves. Finally, the outcome of a coin toss weighted by *Noise* determines if a page

[1]Note that a page may be swapped with a page on its own disk. Such a swap does not affect performance in the steady state, so *Noise* represents the upper limit on the number of changes.

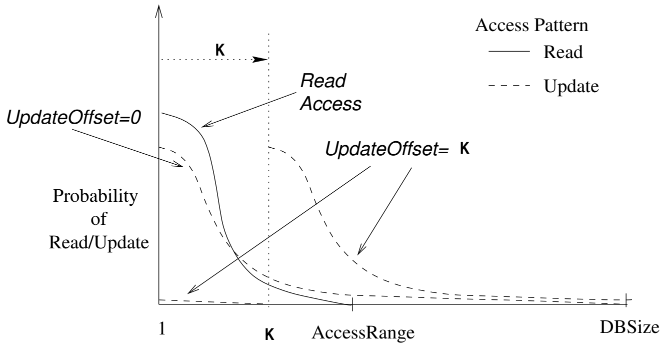


Figure 4.2: Read vs. Update distribution

Parameter	Description
UpdateThinkTime	Time between updates at server (in broadcast units)
UpdateOffset	Update and read access deviation
θ_u	Update Zipf distribution parameter

Table 4.4: Update Parameter Description

is potentially swapped to a new disk. The complete process is shown in Figure 4.1 for the 11-page database in Figure 3.3. The top two rows show the identity function (*Offset = 0*) and the shifting of the four hottest pages to the slowest disk (*Offset = 4*). The third row demonstrates the *Scatter* effect. Finally, the bottom row shows how *Noise* moves pages across different disks.

4.2.2 Server Update Model

The parameters that describe the update model are shown in Table 4.4. For updates to be supported, the parameter *UpdateOn* has to be set. The updates were modeled like in a stock-market database where data is updated at the central server and the clients access the data in a read-only fashion. Updates are generated in the system using a single writer process at the server which simulates the effect of all the updates. The writer process is a simple two-step write-wait loop. After each write access, the writer waits *UpdateThinkTime* broadcast units before making the next update request at the server. The *UpdateThinkTime* parameter determines the rate of updates in the system — the smaller the value, the more frequent the updates.

To successfully capture updates in the broadcast environment, the simulation model has to address the following three issues:

- How often do updates occur (vis a vis read accesses)?
- How does the server notify the clients of these updates?

Parameter	Description
ServerQSize	Length of backchannel queue
PullBW	% of broadcast bandwidth for pulls

Table 4.5: Backchannel Parameter Description

- What is the distribution of updated pages (vis a vis the read access pattern)?

The parameter *UpdateThinkTime* addresses the first question – the larger it is, the more infrequent the updates. In a traditional system, the second question is handled using two techniques — invalidation and propagation messages from the server to the clients. In Chapter 7, which presents the update study, we highlight these techniques and describe how they can be adapted to the broadcast environment.

The simulator uses two parameters θ_u and *UpdateOffset* to address the third problem. We assume that the writer updates pages at the server using a Zipf distribution with parameter θ_u similar to the read access distribution at the client. However, unlike the read access, which is restricted to the first *AccessRange* pages, the update distribution is spread over the complete range 1 to *DBSize*. Unlike reads which are local to a particular client, the updates happen at the server (and are not specific to a single client) and thus, encompass the entire database.

The parameter *UpdateOffset* is used to introduce disagreement between the client access pattern and the update distribution at the server (Figure 4.2). When *UpdateOffset* is 0, the overlap between the two distributions is the greatest, i.e., the client's *hottest* pages (for read access) are also the most frequently updated pages. An *UpdateOffset* of K shifts the update distribution by K pages as shown in Figure 4.2 making the pages which are most often updated be those that are of lesser interest to the client (for read).

4.2.3 Server Backchannel Model

The parameters that are used to model the backchannel at the server are shown in Table 4.5. For the server to support the backchannel, the parameter *BackChannelOn* should be set. Each client is assumed to have a dedicated connection to the server (e.g., via a phone line) and any request sent on it reaches the server instantly. In other words, the cost of communication is assumed to be free. However, once the message reaches the server, there is explicit contention for the server resources. The model of the server with backchannel support is shown in Figure 4.3.

The server queues all the backchannel requests and processes them in a FIFO fashion. The size of this queue is given by the parameter *ServerQSize*. A request is dropped if either the queue is already full or if there is a pre-existing queued request for the page already. We assume that the client gets no feedback from the server about the success or failure of a request.[2]

[2]We assume that clients do not retry dropped requests.

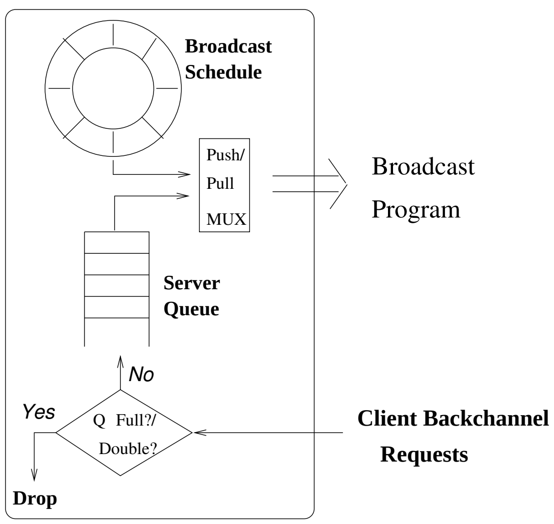


Figure 4.3: Server Backchannel Model

The service rate of the queue is determined by the parameter *PullBW*. *PullBW* determines the percentage of the broadcast slots allocated for pages explicitly pulled by the clients. Before every page is broadcast, a coin weighted by *PullBW* is tossed and depending on the outcome, either the requested page at the head of queue is broadcast or the regular broadcast program continues. Note that the regular broadcast is not interrupted if the server queue is empty and thus, *PullBW* is only an upper limit on the bandwidth used to satisfy backchannel requests. The significance of these parameters will be highlighted again in Chapter 8.

4.3 Experimental Methodology

The bulk of the results in this thesis are based on simulation studies using the model described in this chapter. Simulation is an extremely powerful tool and its utility in exploring the client-server environment has been demonstrated earlier in [Fra93]. As in the case of this thesis, simulations can provide insight into the functioning of new and emerging environments which either lack the technology or are prohibitively expensive to be built in practice. In many systems, inserting probes to gather data implicitly affects the run-time performance, corrupting the utility of the data and defeating the very purpose of the study. In such cases, simulations may be the only feasible outlet for experimentation. Simulations also provide a much larger domain for exploration by allowing system parameters to take on values which may be currently unachievable in practice. However, the downside of simulations is that that a number of simplifying assumptions have to made about the environment in order to be able to design a tractable model and build the simulation software.

The primary goal of the simulation subsystem presented in this thesis is to study the performance of clients operating in a broadcast-based environment under various settings. To that end, the simulator provides a number of parameters which can be tuned. These parameters span the systems resources (e.g., cache size, think time), workload characteristics (e.g., uniform, skewed) and various algorithmic choices made in the design. Given the novel nature of broadcast-based systems, data on system characteristics and usage patterns were not always available. Consequently, we have had to occasionally make "best guess" choices to fill the gaps. However, as much as possible we have tried to mirror the client and server execution and their workload characteristics to earlier studies on client-server database systems [Fra96].

One of the fundamental drawbacks of any simulation based approach is the difficulty in validating the *correctness* of the results. Whereas an error in coding can lead to a fatal crash in a real system, in a simulation, they merely lead to a different output. Consequently, a lot of care needs to be taken to ensure that the code is bug free. In the absence of a real environment to validate the simulation code, the following steps were taken to build confidence in output generated. First, small tests were run and their traces manually checked for errors. Also, when possible, inputs for which the results were known a priori were used to validate the correctness. Secondly, a number of checks were placed in the system to detect inconsistencies both during and at the end of the run. In the experiments which reported the steady-state performance, care was taken to eradicate the effect to cache warm up. For the steady state experiments, the measurement were typically started 4000 accesses after the client cache was full and the runs were stopped only when the response times were within 1% for more than 8,000 accesses. Finally, once the results were tabulated and the conclusions drawn, care was taken to verify these conclusions independently using the supporting data generated during the run. Often, this inference step required adding new probes and re-running the experiments.

The results presented in this thesis are only a small portion of the results generated in the course of the simulations. These results were chosen because they represent many of the unique performance aspects and tradeoffs of the Broadcast Disks environment and because they identify important areas of further study. Due to the simplifications required to make the simulation model tractable and because time and compute resources allowed only a small volume of the simulation space to be explored, these results should be interpreted *qualitatively* and not *quantitatively*. The contributions of these results are not in giving absolute numbers, which however accurate, will become obsolete with advances in technology. Instead, these results should be viewed as highlighting the various trade-offs that determine the system performance and identifying the conditions which make one algorithmic choice more appropriate over the others.

Chapter 5

Demand-Driven Caching

5.1 Introduction

Chapter 3 described the various desirable properties of broadcast programs and highlighted the differences in cost metrics for client caching in a broadcast environment. In this chapter, these characteristics of Broadcast Disks is explored quantitatively using the simulation model. This is be done in two phases. First, the performance of a number of disk configurations is studied for a client which performs no caching. These experiments provide insight into the basic structure and properties of broadcast programs. Then, caching at the client is introduced which raises a number of new issues. The first set of cache-based experiments investigate the performance of traditional page replacement policies. These results highlight the drawbacks of standard caching and motivate the need for cost-based caching. The performance of $\mathcal{PIX}$, an optimal cost-based caching algorithm, is then studied. Finally, the feasibility of implementing cost-based policies in practice is considered. In particular, the performance and robustness of $\mathcal{LIX}$, an implementable approximation to $\mathcal{PIX}$, is investigated.

There are three major issues being addressed in this chapter:

- *Multi-disk Broadcast*. What is the performance benefit of a multi-disk broadcast over a flat broadcast program for both caching and non-caching clients?
- *Client Caching*. Caching raises two key issues:
  - ❖ What is the impact of the cache at the client on the organization of the broadcast program?
  - ❖ What is the performance improvement due to caching?
- *Server Program Variance*. The server generates a broadcast program tailored to the needs of the entire client population. How sensitive is a particular client's responsiveness to variations in the server broadcast (as determined by *Noise*)?

The results in this chapter are based on some assumptions. These assumptions supplement those presented in Section 3.4 and are as follows:

- *Broadcast program is static.* The client population and their access patterns do not change. This implies that the content and the organization of the broadcast program remains static.
- *Read-only access.* There are no updates either by the clients or at the servers.
- *On-demand access.* Clients retrieve data items from the broadcast on demand; there is no prefetching.
- *No backchannel.* Clients make no use of their upstream communications capability, i.e., they provide no feedback to servers.

Table 5.1 shows the parameter settings used in this chapter. The primary performance metric employed in this study is the response time at the client, measured in *Broadcast Units*. The server database size (*ServerDBSize*) was 5000 pages, and the client access range *AccessRange* was 1000 pages. The client accesses the pages in the access range using a Zipf distribution with parameter 0.95, as described in Section 4.1. Only a single client $\mathcal{MC}$, the measure client was simulated. The client cache size was varied from 1 (i.e., no caching) to 500 (i.e., half of the access range). To get a better understanding of the tradeoffs a number of different two-disk and three-disk configurations were studied.

5.2 Performance of Cache-less Clients

5.2.1 Experiment 1: No Caching, Basic Broadcast Tradeoffs

The first set of results examine the case where the client performs no caching (i.e., it has a cache size of one page). Figure 5.1 shows the client response time vs. Δ for a number of two and three disk configurations. In this graph, *Noise* is set to 0%, i.e., the server is providing preferential treatment to the client (i.e., it is giving highest priority to this client's pages). As Δ is increased along the x-axis of the figure, the skew in the relative speeds of the disks is increased (as described in Section 4.2.1). As shown in the figure, the general trend in these cases is that response time improves with increasing disk skew. When $\Delta = 0$, the broadcast is flat (i.e., all disks rotate at the same speed). In this case, as would be expected, all disks result in a response time of 2500 pages — half the *ServerDBSize*. As Δ is increased, all of the disk configurations shown provide an improvement over the flat disk. This demonstrates the utility of the multi-disk broadcast program for a client which has a skewed access to the database.

PrefetchOn	False
UpdateOn	False
BackChannelOn	False
NumClients	1
ThinkTime	2.0
ServerDBSize	5000
AccessRange	1000
CacheSize	50(5%), 250(25%), 500(50%)
Δ	1,2,. . .7
θ	0.95
Offset	0, *CacheSize*
Noise	0%, 15%, 30%, 45%, 60%, 75%
RegionSize	50

Table 5.1: Basic Parameter Settings

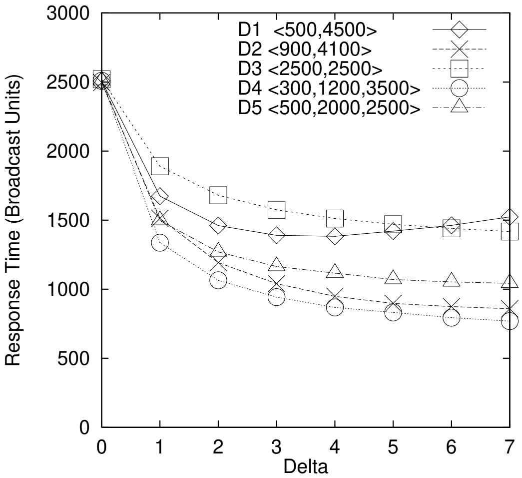


Figure 5.1: Client Performance, *CacheSize* = 1, *Noise* = 0%

Turning to the various disk configurations, we first examine the two-disk configurations: D1, D2, and D3. For D1, 500 pages fit on the first (i.e., fastest) disk. Because *Noise* and *Offset* are both zero, the hottest half of the client's access range is on the fast disk, and the colder half is on the slower disk. As Δ is increased, performance improves until $\Delta = 3$ because the hotter pages are brought closer. Beyond this point, the degradation caused by the access to the slow pages (which get pushed further away) begins to hurt performance. In contrast, D2, which places 90% of the client access range (900 pages) on the fast disk improves with increasing Δ for all values of Δ in this experiment. Because most of the accessed pages are on the fast disk, increasing Δ pushes the colder and unused pages further away, allowing the accessed pages to arrive more frequently. At some point, however, the penalty for slowing down the 10% will become so great that the curve will turn up again as in the previous case. The final two-disk configuration, D3, has equal sized disks. Although all of the accessed data fits on the fast disk, the fast disk also includes many unaccessed pages. The size of the fast disk causes the frequencies of the pages on this disk to be lower than the frequencies of pages on the fast disks of D2 and D1 at corresponding values of Δ. As a result, D3 has the worst performance of the two-disk configurations for most of the Δ values shown.

Turning to the three-disk configurations: D4 and D5, it can be seen that configuration D4, which has a fast disk of 300 pages has the best performance across the entire range. At a Δ of 7, its response time is only one-third of the flat-disk response time. D5, which is simply the D3 disk with its first disk split across two disks, performs better than its two-disk counterpart. The extra level of disk makes it easier to match the broadcast program to the client's needs. However, note that response time for D5 is typically higher than the two-disk D2, and thus, the extra disk level does not necessarily ensure better performance.

5.2.2 Experiment 2: No Caching, Noise Sensitivity

In the previous experiment, the broadcast program generation was done giving the simulated client's access pattern the highest priority. This experiment examines the performance as the server shifts its priority away from this client (i.e., as *Noise* is increased). These results are shown in Figures 5.2 and 5.3, which show how the client performs in the presence of increasing noise for configurations D3 (two-disks) and D4 (three-disks) from Figure 5.1 respectively. As expected, performance suffers for both configurations as the *Noise* is increased; as the mismatch between the broadcast and the client's needs increases, the skew in disk speeds starts to hurt performance. Ultimately, if the mismatch becomes great enough, the multi-disk approach can have worse performance than the flat disk. This is shown in the performance disk of D3 (Figure 5.2). This susceptibility to a broadcast mismatch is to be expected, as the client accesses all of its data from the broadcast channel. Thus, it is clear that if a client does not have a cache, the broadcast must be well suited for that client's access demands in order to gain the benefits of the multi-disk approach. Also, as the graph shows, the multi-disk program can be a double-edged sword — while it can offer tremendous performance improvement when the program is designed well, it can also degrade responsiveness well below a flat disk in the worst case.

5.3 Performance of Standard Caching

The results of the last section demonstrated that the use of the multi-level disk can significantly improve performance for a cache-less client. However, the performance can suffer if the broadcast is not tailored to the client's request pattern. In this section, a client cache is introduced to reduce the expected page access delay and to increase the client's tolerance to

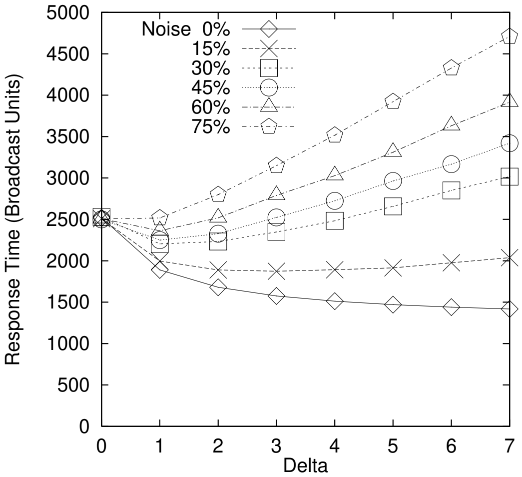


Figure 5.2: Noise Sensitivity
Disk D3(<2500,2500>), *CacheSize* = 1

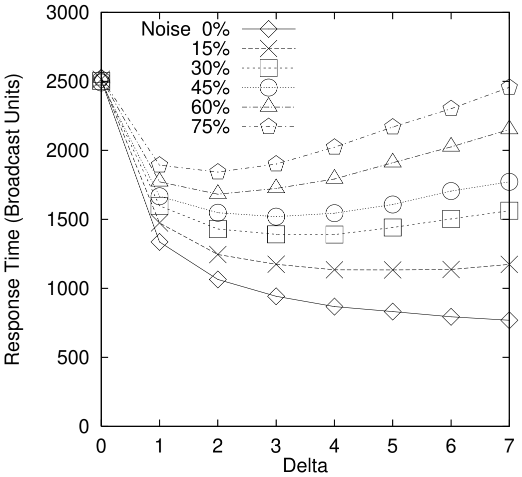


Figure 5.3: Noise Sensitivity
Disk D4(<300,1200,3500>), *CacheSize* = 1

mismatches in the broadcast program. The performance of the caching client is studied using $\mathcal{P}$ (described next), a traditional probability-based page replacement algorithm. As the results show, caching helps a long way in mitigating the effects of server program variance. However, as pointed out in Section 3.3.2, the role of caching is fundamentally different in this environment and the failure of the standard caching algorithms to exploit the characteristics of the broadcast program leads to its lack of scalability.

5.3.1 $\mathcal{P}$: A Standard Page Replacement Algorithm

$\mathcal{P}$ is an idealized page replacement policy to do demand-driven caching. $\mathcal{P}$ attempts to always keep the pages in the cache which have the highest probability of access. So, in response to a page miss, $\mathcal{P}$ chooses as the victim the cache-resident page with the lowest probability of access. $\mathcal{P}$ is a pure strategy in that it requires perfect knowledge of probability of access of all pages. It is used in these studies for two reasons. One, it is trivial to implement in the simulator since the probability of each page is known a priori. Two, being a very straightforward algorithm, it provides a better understanding of the environment in a simplified setting and serves as a baseline for comparison to other (implementable) policies. The popular caching algorithms like LRU and LFU [Tan92] are implementable variants of $\mathcal{P}$.

5.3.2 Experiment 3: Influence of Cache on Server Program

In the no caching case, the client performs best when there is a perfect match with the broadcast, i.e., when the server fills its disks from the fastest to the slowest with the client's pages

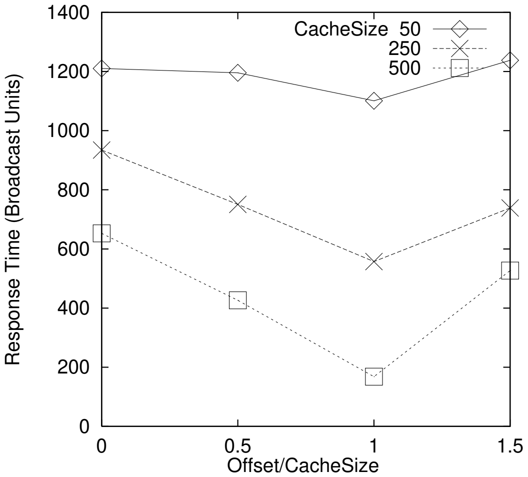


Figure 5.4: *Offset* Sensitivity
Disk D3(<2500,2500>), Δ = 3

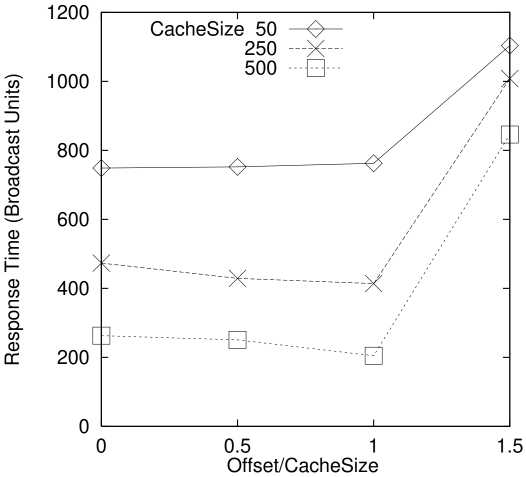


Figure 5.5: *Offset* Sensitivity
Disk D4(<300,1200,3500>), Δ = 3

in decreasing order of probability of access. Alternatively said, the best broadcast for a client with no cache is at an *Offset* (described in Section 4.2.1) of zero. The introduction of a client cache, however, changes this. A page replacement policy tends to favor a subset of pages over others; in steady state, therefore, some pages are more likely to be cache-resident. Such "hot" pages, are *less* likely to be obtained from the broadcast and thus, should be broadcast infrequently. Assuming that clients choose to cache pages based on their probability of access, then the best broadcast will be obtained with a non-zero *Offset*. A non-zero *Offset* implies that the server broadcasts a portion of the client's hottest pages on the slowest disk and fills the faster disks with the remainder of the client's access range.

Figures 5.4 and 5.5 show the response time of the client for varying offset when the two-disk D3, and the three-disk D4 are used (in this case, with $\Delta = 3$), respectively. Each graph shows the results for three different cache sizes under the $\mathcal{P}$ page replacement policy. The x-axis gives the ratio of the *Offset* to the *CacheSize*.

The two-disk D3 (see Figure 5.4) shows the most sensitivity to *Offset*. In the figure, the best response time occurs when the *Offset* is equal to the client cache size. Because the $\mathcal{P}$ policy retains the *CacheSize* hottest pages, these pages should be pushed to the slower disk, which is exactly what an *Offset* of size *CacheSize* does. If the *Offset* is too small then the server wastes a significant portion of the fastest disk for pages already in the client's cache. If it is too large, then more hot pages than will fit in the client's cache are pushed to the slowest disk resulting in huge penalties for cache misses on these pages. The three-disk configuration (D4) (Figure 5.5), is not as sensitive to an undersized *Offset* because the middle disk provides a buffer between the fast and slow disks for those pages that should have been on the fast disk. However, it is

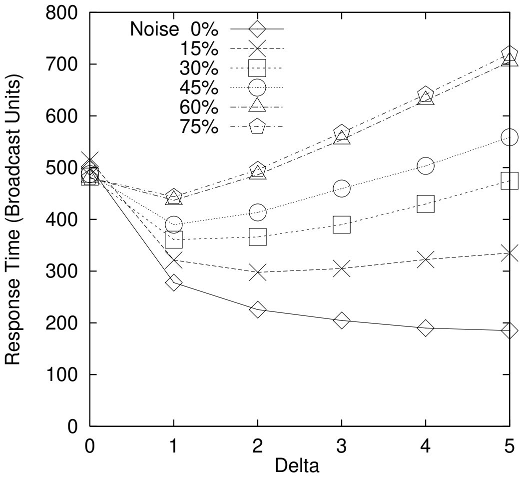


Figure 5.6: Noise Sensitivity, $\mathcal{P}$ Policy
Disk D4, *CacheSize* = 500

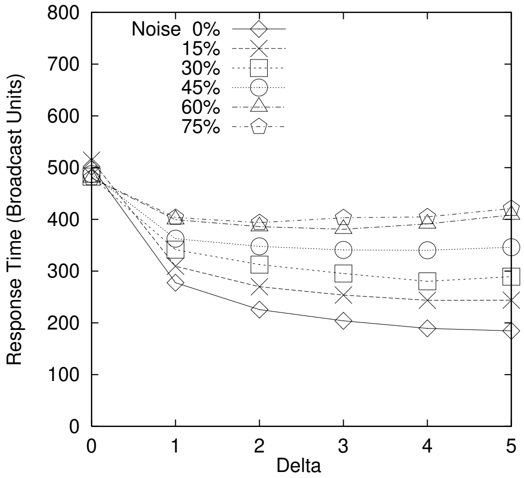


Figure 5.7: Noise Sensitivity, $\mathcal{PIX}$ Policy
Disk D4, *CacheSize* = 500

much more sensitive to an oversized *Offset*, as the differential between the fast and slow disks is greater, making cache misses on the slow disk more expensive.

The two graphs show that the value of *Offset* providing the best performance is *CacheSize*. Given the caching policy $\mathcal{P}$, it can be easily argued that this is always the ideal *Offset*. $\mathcal{P}$, by definition, retains the *CacheSize* hottest pages in the cache. Consequently, these pages should be pushed to the slower disk, which is exactly what an *Offset* of size *CacheSize* does. If the *Offset* is too small then the server wastes a significant portion of the fastest disk for pages already in the client's cache. If it is too large, then more hot pages than will fit in the client's cache are pushed to the slowest disk resulting in huge penalties for cache misses on these pages. Thus, for the remainder of the experiments in this chapter, the server will generate a broadcast with an *Offset* of *CacheSize*.

5.3.3 Experiment 4: $\mathcal{P}$, Noise Sensitivity

As argued in the last section, the best performance for a client using the $\mathcal{P}$ policy is when the server broadcast has an *OffSet* equal to the size of the client cache. Given this *Offset*, the effectiveness of a cache in allowing a client to tolerate *Noise* in the broadcast is investigated in this experiment. Figure 5.6 shows the impact of increasing *Noise* on the performance of the three-disk configuration D4 as Δ is varied. In the case shown, *CacheSize* and *Offset* are both set to 500 pages. Comparing these results with the results obtained in the no caching case (Figure 5.3), we see that although as expected the cache greatly improves performance in an absolute sense, surprisingly, the cache-based numbers are if anything, somewhat *more* sensitive to the degree of *Noise* than the non-caching numbers. For example, in the caching

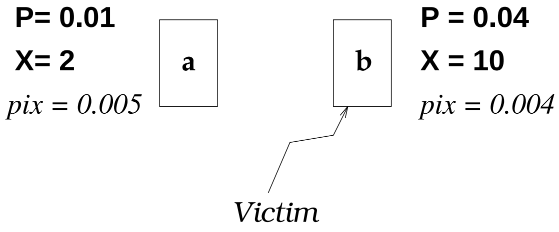


Figure 5.8: Page Replacement in $\mathcal{PIX}$

case, when Δ is greater than 2, the higher degrees of noise have multi-disk performance that is worse than the flat disk performance, whereas this crossover did not occur for similar Δ values in the non-caching case. The reason for this additional sensitivity is that when *Noise* is low and *Offset* = *CacheSize*, $\mathcal{P}$ does exactly what it should do — it caches those hot pages that have been placed on the slowest disk, and it obtains the remainder of the hottest pages from the fastest disk. However, as noise increases, $\mathcal{P}$ caches the same pages regardless of what disk they are stored on. Caching a page that is stored on the fastest disk is often not a good use of the cache, as those pages are broadcast frequently. As noise increases, $\mathcal{P}$'s cache hit rate remains the same, but its *cache misses become more expensive*, as it has to retrieve some pages from the slower disks. These expensive cache misses are the cause of $\mathcal{P}$'s sensitivity to *Noise*. Thus, in this environment, the cost of a miss is as important as trying to improve the hit rate. This is the motivation for $\mathcal{PIX}$, a cost-based caching algorithm described next.

5.4 Performance of Cost-based Caching

5.4.1 $\mathcal{PIX}$: A Cost-based Caching Algorithm

Section 3.3.3 described the notion of cost-based caching and motivated why the traditional approach of probability-based caching is poorly suited in the broadcast environment. The drawbacks were also demonstrated quantitatively in the experiments in the last section. $\mathcal{PIX}$ is a cost-based algorithm for demand-driven caching.

Let p_i be the probability of access and x_i be the frequency of broadcast (i.e., number of occurrences per broadcast period) of page i. For each cache resident page i, $\mathcal{PIX}$ assigns a *pix* value which is the ratio p_i/x_i. In response to a miss, $\mathcal{PIX}$ chooses the page with the *lowest pix* value as the replacement victim. The motivation for *pix* based replacement is that the victim should be the page which has the lowest local utility (i.e., lowest access probability) *and* the lowest re-acquisition cost (i.e., highest broadcast frequency). In fact, this ratio can be shown to be optimal in minimizing the average delay. An optimality proof is outlined in Appendix A.

Figure 5.8 shows page replacement using $\mathcal{PIX}$ for two pages a and b. $\mathcal{PIX}$ chooses b as a victim because it has a lower *pix* value whereas $\mathcal{P}$ would choose a because it has a lower probability of access.

The $\mathcal{P}$ replacement policy was found to be sensitive to noise because it ignored the *cost of re-acquiring a page* when choosing a victim for replacement. $\mathcal{PIX}$ addresses this deficiency by using the frequency of broadcast as a metric to estimate re-acquisition cost during page replacement.

5.4.2 Experiment 5: $\mathcal{PIX}$, Noise Sensitivity

Figure 5.7 shows the response time of the client using $\mathcal{PIX}$ for the same simulation space as the last experiment (Figure 5.6), which gave the performance of $\mathcal{P}$. Comparing the two figures it can be seen that $\mathcal{PIX}$ is much more successful at insulating the client response time from effects of *Noise*. Of course, an increase in *Noise* still results in a degradation of performance; this is to be expected. However, unlike the case with $\mathcal{P}$, using $\mathcal{PIX}$ the performance of the client remains better than the corresponding flat disk performance for all values of *Noise* and Δ in this experiment. Under $\mathcal{PIX}$, the performance of the client for a given *Noise* value remains stable as Δ is increased beyond a certain point. In contrast, under $\mathcal{P}$, in the presence of noise, the performance of the client quickly degrades as Δ is increased beyond a value of 1 or 2. This experiment demonstrates the potential of cost-based replacement for making the Broadcast Disks approach practical for a wider range of applications.

Figure 5.9 shows results from the same set of experiments from a different viewpoint. In this figure, the effect of increasing noise on the response time of the two algorithms for $\Delta = 3$ and $\Delta = 5$ is shown. The performance for the flat disk ($\Delta = 0$) is given as a baseline.[1] Note that $\mathcal{P}$ degrades faster than $\mathcal{PIX}$ and eventually becomes worse than the flat disk at around *Noise* = 45%. $\mathcal{PIX}$ rises gradually and manages to perform better than the flat disk within these parameters. Also, notice how $\mathcal{P}$'s performance degrades for $\Delta = 5$; unlike $\mathcal{PIX}$ it fails to adapt the cache contents with increasing differences in disk speeds.

The performance differences between the two algorithms result from the differences in where they acquire their pages from. Figure 5.10 shows a histogram of the location of hits for the two algorithms. The figure is for the 3-disk broadcast D4, a cache of 500 pages and for *Noise = 30%*. It is interesting to note that $\mathcal{PIX}$ has a lower cache hit rate than $\mathcal{P}$. A lower cache hit rate does not mean lower response times in broadcast environments; the key is to reduce expected latency by caching important pages that reside on the slower disks. This is a fundamental difference from caching in traditional systems. $\mathcal{PIX}$ gets fewer pages from the slowest disk than does $\mathcal{P}$, even though it gets more pages from the first and second disks. In this case, this tradeoff results in a net performance win.

[1] Note that at $\Delta = 0$ (i.e., a flat disk), $\mathcal{P}$ and $\mathcal{PIX}$ are identical, as all pages are broadcast at the same frequency.

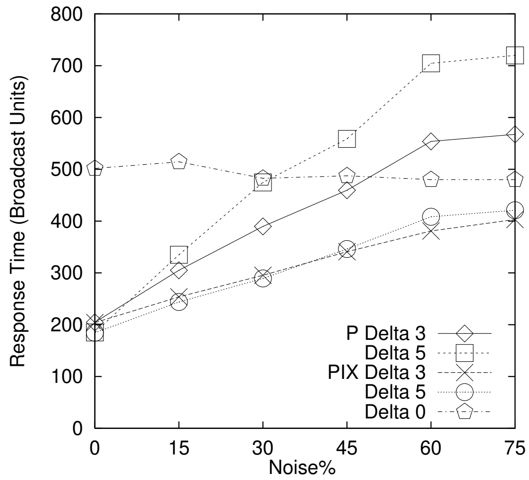


Figure 5.9: $\mathcal{P}$ vs. $\mathcal{PIX}$ With Varying Noise, Disk D4, *CacheSize* = 500

5.4.3 $\mathcal{LIX}$: Implementable Cost-based Caching

The previous sections have shown that multi-disk broadcast environments have special characteristics which when correctly exploited can result in significant performance gains. They also demonstrated the need for cost-based page replacement and examined a cost-based algorithm ($\mathcal{PIX}$). Unfortunately, like $\mathcal{P}$, the policy on which it is based, $\mathcal{PIX}$ is not an implementable algorithm. However, based on the insight that gained by examining $\mathcal{P}$ and $\mathcal{PIX}$, this section presents the design and performance of $\mathcal{LIX}$, an implementable approximation for $\mathcal{PIX}$.

$\mathcal{LIX}$ is a modification of LRU[Tan92] that takes into account the broadcast frequency. LRU maintains the cache as a single linked-list of pages. When a page in the cache is accessed, it is moved to the top of the list. On a cache miss, the page at the end of the chain is chosen for replacement.

In contrast, $\mathcal{LIX}$ maintains a number of smaller chains: one corresponding to each disk of the broadcast ($\mathcal{LIX}$ reduces to LRU if the broadcast uses a single flat disk). A page always enters the chain corresponding to the disk in which it is broadcast. Like LRU, when a page is hit, it is moved to the top of *its own* chain. When a new page enters the cache, $\mathcal{LIX}$ evaluates a *lix* value (see next paragraph) only for the page at the bottom of each chain. The page with the smallest *lix* value is ejected, and the new page is inserted in the appropriate queue. Because this queue might be different than the queue from which the slot was recovered, the chains do not have fixed sizes. Rather, they dynamically shrink or grow depending on the access pattern at that time. $\mathcal{LIX}$ performs a constant number of operations per page replacement (proportional to the number of disks) which is the same order as that of LRU. Figure 5.11 shows an example of $\mathcal{LIX}$ for a two-disk broadcast. Pages `g` and `k` are at the bottom of each

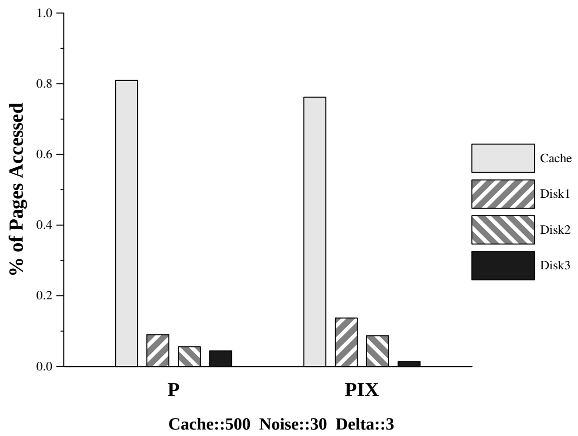


Figure 5.10: Access Locations for $\mathcal{P}$ vs. $\mathcal{PIX}$

Disk D4, *CacheSize* = 500, Noise = 30%, Δ = 3

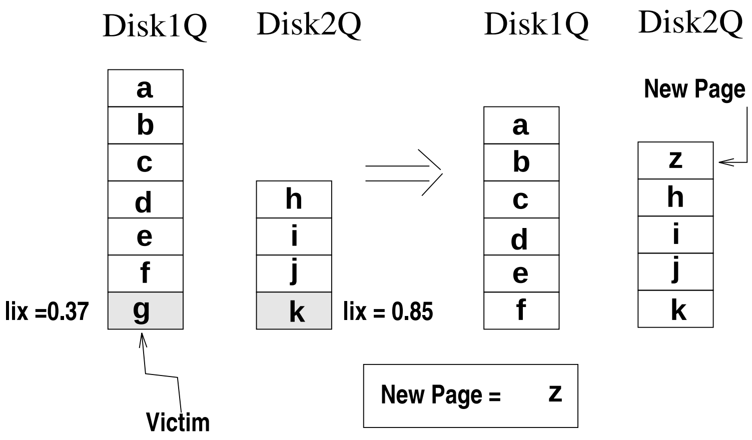


Figure 5.11: Page Replacement in $\mathcal{LIX}$

chain. Since g has a lower *lix* value it is chosen as the victim. The new page z, being picked from the second disk, joins Disk2Q. Note the relative changes in the sizes of both the queues.

The idea behind $\mathcal{LIX}$ is as follows. $\mathcal{LIX}$ (like $\mathcal{PIX}$) has to determine the page with the lowest probability of access to frequency of broadcast ratio at every replacement. Since the possible values of broadcast frequency are equal to the number of disks, by inserting a page into the queue corresponding to its disk, $\mathcal{LIX}$ automatically sorts pages by their broadcast frequency. To find the page with the lowest ratio, $\mathcal{LIX}$ only needs to find the page with the lowest probability of access in each chain and compare their ratios to determine the victim. To find this candidate victim in each chain, $\mathcal{LIX}$ uses the rationale on which LRU is based – lower down the chain, lower the probability of access. So, it suffices for $\mathcal{LIX}$ to compare the *lix* metric (described next) for just the pages in the bottom of each chain to choose a victim.

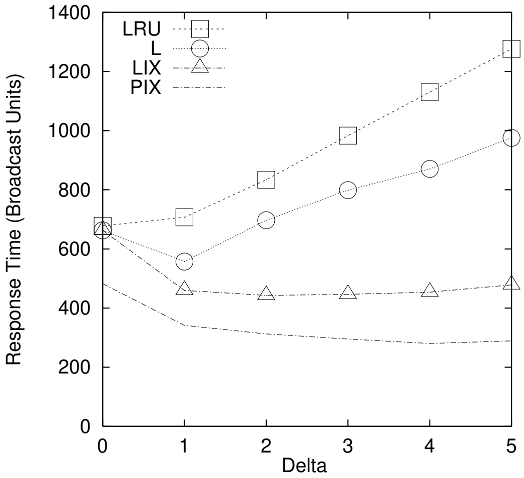


Figure 5.12: Sensitivity to Δ - Disk D4

CacheSize = 500, Noise = 30%

In order to compute the *lix* value, the algorithm maintains two data items per cached page i : a running probability estimate (p_i) and the time of the most recent access to the page (t_i). When the page i enters a chain, p_i is initially set to zero and t_i is set to the current time. If i is hit again, the new probability estimate for i is calculated using the following formula:

$$p_i = \Lambda / (CurrentTime - t_i) + (1 - \Lambda) p_i.$$

t_i is then subsequently updated to the current time. Λ is a constant used to appropriately weigh the most recent access with respect to the running probability estimate; in these experiments, it is set to 0.25. Λ is restricted to be between 0 and 1 and thus, p_i also has the same range.[2] This formula is evaluated for the least recently used pages of each chain to estimate their current probability of access. This value is then divided by the frequency for the page (which is known exactly) to get the *lix* value. The page with the lowest *lix* value is ejected from the cache. $\mathcal{LIX}$ is a simple approximation of $\mathcal{PIX}$, yet it performs surprisingly well (as is shown below). Better approximations of $\mathcal{PIX}$, however, might be developed using some of the recently proposed improvements to LRU like 2Q[JS94] or LRU-k[OOW93].

5.4.4 Experiment 6: $\mathcal{LIX}$ vs. LRU

The next set of experiments are similar to those for $\mathcal{P}$ and $\mathcal{PIX}$ and compare $\mathcal{LIX}$ and LRU. However, unlike $\mathcal{P}$, the best performance for LRU isn't at an offset equal to the cache size. Being only an approximation of $\mathcal{P}$, LRU isn't able to retain all of the hot pages that are stored

[2] A similar hysteresis formula is used to estimate the round-trip time for a TCP connection [Ste94].

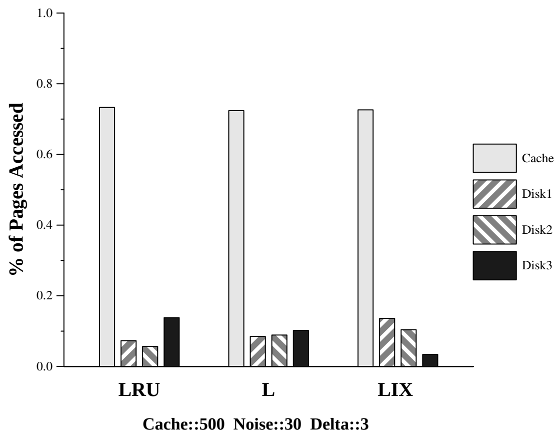


Figure 5.13: Page Access Locations - Disk D4

CacheSize = 500, Noise = 30%, Δ = 3

on the slowest disk and thus, it performs poorly at this offset. For similar reasons, $\mathcal{LIX}$ also does not perform best at this offset. As a result, we also compared the performance of $\mathcal{LIX}$ and LRU to a modified version of $\mathcal{LIX}$ called $\mathcal{L}$. $\mathcal{L}$ behaves exactly like $\mathcal{LIX}$ except that it assumes the same value of frequency of broadcast for all pages. Thus, the difference in performance between $\mathcal{L}$ and LRU indicates how much better (or worse) an approximation of probability $\mathcal{L}$ provides over LRU, and the performance difference between $\mathcal{LIX}$ and $\mathcal{L}$ shows the role that broadcast frequency plays (if any) in the performance of the caching strategies.

Figure 5.12 shows the performance of the three algorithms for different values of Δ. These results show the sensitivity of the algorithms to changing Δ for the same case as in Figure 5.9 (i.e., *Offset*=*CacheSize*=500), with *Noise* set to 30%. In this experiment, LRU performs worst and consistently degrades as Δ is increased. $\mathcal{L}$ does better at $\Delta = 1$ but then degrades. The benefits of using frequency are apparent from the difference in response time between $\mathcal{LIX}$ and $\mathcal{L}$. The response time of $\mathcal{LIX}$ is only between 25% to 50% that of $\mathcal{L}$. The solid line on the bottom of the graph shows how the ideal policy ($\mathcal{PIX}$) performs; it does better than $\mathcal{LIX}$, but only by a small margin. The factors underlying these results can be seen in Figures 5.13, which shows the distribution of page access locations for the results of Figure 5.12 when Δ is set to 3. In this case, $\mathcal{LIX}$ obtains a much smaller proportion of its pages from the slowest disk than do the other algorithms. Given that the algorithms have roughly similar cache hit rates, the differences in the distributions of access to the different disks is what drives the performance results here.

Figure 5.14 shows the performance of the three algorithms with varying *Noise* with $\Delta = 3$. In this case, it can be seen that $\mathcal{L}$ performs only somewhat better than LRU. The performance of $\mathcal{LIX}$ degrades with noise as expected, but it outperforms both $\mathcal{L}$ and LRU across the entire region of *Noise* values. These results demonstrate that the frequency-based heuristic of $\mathcal{LIX}$

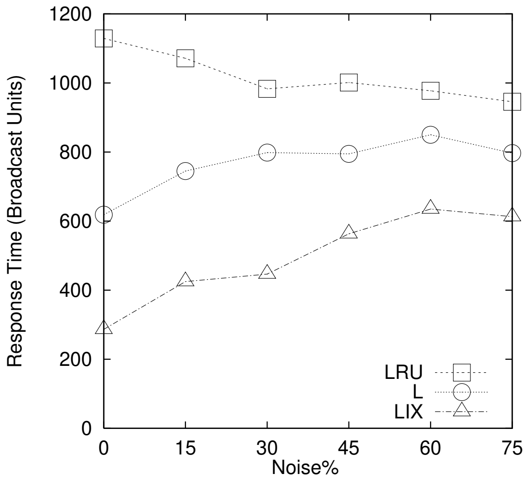


Figure 5.14: $\mathcal{LIX}$ Noise Sensitivity - Disk D4

CacheSize = 500, Δ = 3

can provide improved performance in the presence of noise.

5.5 Conclusions

In this chapter, we evaluated the benefits of a multi-disk broadcast on the client performance and investigated the notion of demand-driven caching in a broadcast framework. The utility of the multi-disk broadcast was studied for both the cacheless clients and the caching clients. The multi-disk broadcast was found to outperform the flat broadcast for a cacheless client with skewed access to the database justifying the intuition behind the Broadcast Disks paradigm. However, the experiments also confirmed the sensitivity of the clients without any cache to variance in the broadcast program as was argued in Section 3.3.2. While a multi-disk program gives significant performance improvement in the best case, the responsiveness can degrade significantly if the program is not tailored to the client access needs.

Introduction of a cache at the client was found to help mitigate the effects of a poorly designed broadcast. However, the traditional probability-based caching failed to fully exploit the characteristics of the multi-disk paradigm. Consequently, while in general, the cache speeded up the client processing, in some cases, it only made the client more sensitive to the vagaries of the broadcast program.

$\mathcal{PIX}$, a cost-based caching algorithm was then introduced and was shown to not only provide better client performance but also better insulate the client from the server broadcast. The notion of cost-based caching, introduced in Section 3.3.3, states that along with the probability of access, the cost of re-acquisition should be accounted for during page replacement. $\mathcal{PIX}$

uses the frequency of broadcast as a measure of the re-acquisition cost to derive an optimal replacement heuristic. Finally, the design and implementation of $\mathcal{LIX}$, a constant time approximation for $\mathcal{PIX}$ based on LRU was described. In spite of a straightforward design, $\mathcal{LIX}$ was found to be a good match for $\mathcal{PIX}$.

Chapter 6

Prefetching from Broadcast Disks

6.1 Introduction

This chapter explores the benefits of prefetching in a broadcast environment. Using prefetching, clients bring pages into the cache before they are accessed so as to minimize the delay when the actual request is made. As pointed out in Section 3.3.4, this environment is ideally suited for prefetching because it presents new pages to the client continually obviating the need for explicit prefetch requests to the server.

Prefetching in a broadcast environment is a very time-critical task. When doing so, the client has to sample every page broadcast and determine if the page is *prefetch-worthy*. In other words, if there exists a cache-resident page which has a lower *utility of caching* than the currently broadcast page, the broadcast page should take its cache slot. If not, the client repeats this process on the next page transmitted. The decision to prefetch a page (or not) needs to be time-efficient. On average, it should be done in time less than it takes to broadcast a page. Otherwise, the client will fail to sample every page.

6.2 $\mathcal{PT}$: A Prefetching Cache Management Strategy

The task of every prefetching algorithm is to efficiently determine the *utility of caching* the currently broadcast page vis a vis the cache-resident pages. As the Tag-team example in Section 3.3.4 demonstrates, a page is most valuable when it is farthest away from the client since a miss then causes the greatest delay. Any metric used to rank the utility of various pages, should capture this observation. In this chapter, we study the performance of a prefetching algorithm called $\mathcal{PT}$.

$\mathcal{PT}$ is a simple prefetching heuristic which for every page takes:

p : the probability of access of the page

t : the time elapsed before the next broadcast for the page

to generate a metric called the pt value of the page by taking the product of p and t. $\mathcal{PT}$ finds the page in the cache with the lowest pt value, and replaces it with the currently broadcast page if the latter has a higher pt value. The t component of the pt metric captures the Tag-team intuition – higher the time until next arrival, more valuable the page. For the same reason, pt values are dynamic — they change with every "tick" of the broadcast. $\mathcal{PT}$ uses the same eviction strategy for both prefetched pages and pages accessed on demand. Thus, it is an *integrated* [CL95] caching and prefetching strategy.

[TC97] describes a class of optimal prefetching algorithms based on lookahead and $\mathcal{PT}$ is shown to be the base class of the hierarchy. $\mathcal{PT}$ is a single-step lookahead algorithm, i.e., it determines the utility of caching a page using the instantaneous cost of the pt metric. The generalization is to lookahead a few clock "ticks" and use the averaged pt value over the interval. The motivation for a longer lookahead is to eliminate transient spikes in the pt values and thus, have a better estimation of the utility. However, while a longer lookahead produces better performance, especially for irregular broadcasts, it comes at the expense of added complexity. The details of the optimality proof of $\mathcal{PT}$ is beyond the scope of this thesis and the interested reader is referred to [TC97].

6.2.1 Static vs. Dynamic Caching Metrics

$\mathcal{PT}$ uses a dynamic caching metric because the pt value of a page is continually changing because of the time to next arrival parameter. On the other hand, the demand-driven algorithm $\mathcal{PIX}$, introduced in the last chapter, uses a static metric for page replacement. If a client's probability of access of a page (P) never changes and the broadcast program is static (i.e., frequency of broadcast (X) remains the same), the *pix* (P/X) value of a page is constant. In this section, we will investigate the tradeoffs between using a static and a dynamic metric.

First, consider how the pt value changes over time. When a page is broadcast, t is highest since ignoring the page at that point puts the client at a maximal distance from the page. In other words, the moment that a page passes by is the moment at which the wait to get it again is the longest. From that point on, the time parameter steadily decreases until subsequent rebroadcast of the page, thus creating a sawtooth function of time for $\mathcal{PT}$. Figure 6.1 traces this saw tooth effect for three pages A, B and C. The access probabilities and broadcast frequencies for the three pages are given in Table 6.1. Consider time unit 5. At this point, page A is broadcast and its pt value shoots up to its peak. A similar effect happens for B at time units 2, 12, 22,$\cdots$; however, its maximum pt value is only half that of A because of its lower access

Page	Access Probability	Broadcast Frequency (per Period)	Repetition Period	*pix* Value
A	P	2	10 units	P/2
B	P/2	2	10 units	P/4
C	P/2	1	20 units	P/2

Table 6.1: Access and Broadcast Frequencies for Figure 6.1

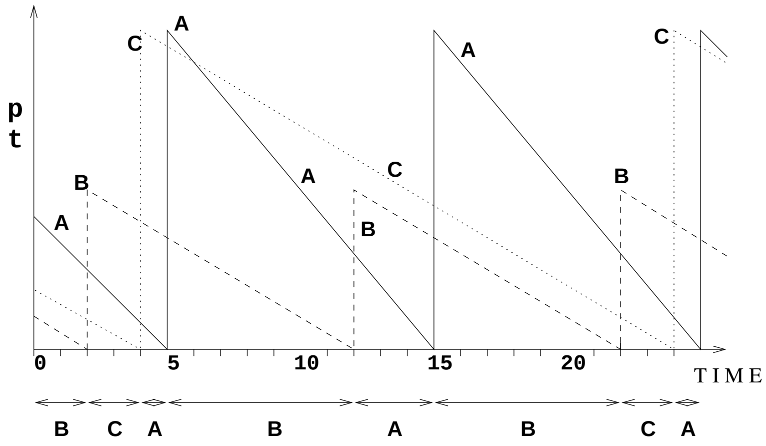


Figure 6.1: pt vs. Time

probability. C, which has the same access probability as B, equals the maximum pt value (at time units 4, 24, 44,$\cdots$) for A because of its longer broadcast period. The line along the bottom of the figure shows the potential replacement victim at each time unit. Table 6.1 also shows the *pix* values for the three pages. Since B has the lowest *pix* value, $\mathcal{PIX}$ would always choose B as a victim over A or C. However, as seen in Figure 6.1, unlike $\mathcal{PIX}$, there are time intervals when $\mathcal{PT}$ would choose C or even A as a replacement victim over B. This is a consequence of the dynamic nature of the pt metric.

$\mathcal{PIX}$ uses the *average cost of access* when determining the worth of caching a page. In other words, for the example in Table 6.1, on the average, there is a greater utility of caching pages A and C over page B. Thus over time, pages with high *pix* values start settling into the cache. Eventually, in steady state, only one cache slot has any replacement activity – the other (cache size minus one) slots have the highest *pix* valued pages which are never evicted by $\mathcal{PIX}$.

$\mathcal{PT}$, on the other hand, uses the *instantaneous cost of access* in its caching decision. Thus, even though, A and C are more valuable on the average than B, there are portions of the broadcast cycle when they have a lower utility of caching. Due to this saw-tooth behavior, items are most sticky right after they are brought into the cache and $\mathcal{PT}$ will tend to drop high probability items just before they are re-broadcast. $\mathcal{PT}$ is in effect adjusting the allocation of cache slots to pages so that (subject to cache size) they are likely to be in the cache during those portions of the broadcast cycle when they would be most expensive to acquire from the broadcast. Note that this effect of cycling pages in and out of the cache is suggestive of Tag-team caching (Section 3.3.4). In fact, the results of the Section 6.3.2 show that $\mathcal{PT}$ is able to achieve the performance of the Tag-team example described in that section. Thus, unlike $\mathcal{PIX}$, no page is guaranteed a cache slot for the entire life of the broadcast by $\mathcal{PT}$.

From a theoretical standpoint, the area under the $\mathcal{PT}$ curve in Figure 6.1 for each page, would be the same as the area under a $\mathcal{PIX}$ curve, if one were similarly plotted (it would be a straight line parallel to the time axis). Mathematically, the correlation of $\mathcal{PT}$ and $\mathcal{PIX}$ is as follows:

Let for some page p,

$[\texttt{pt}]_p^t$: pt value of page p at time unit t.

$[pix]_p$: *pix* value of page p.

C : Length of a complete broadcast cycle

Assuming, a static broadcast,

$$\sum_{t=1}^{C} [\texttt{pt}]_p^t = C \times [pix]_p.$$

It should be noted that any prefetch scheme is potentially very expensive to implement. A demand-driven caching scheme has to sample the broadcast only in response to a miss. A prefetch scheme, however, needs to sample the broadcast continually. Also, if the metric is time-based like in $\mathcal{PT}$, it needs to be re-calculated for all the cache-resident pages at every tick making it very expensive. Thus, while $\mathcal{PT}$ is used as an ideal case in this initial experiments in this chapter, an implementable approximation to $\mathcal{PT}$, called $\mathcal{APT}$, is investigated in Section 6.4.

6.3 Experiments and Results

6.3.1 Parameter Settings

Table 6.2 shows the parameter settings for the prefetch experiments. The server database size (*ServerDBSize*) was 3000 pages and the client access range (*AccessRange*) was 1000 pages. The client cache size was varied from 1 (i.e., no caching) to 1000 (i.e., the entire access range). The client uses both the uniform and the zipf access distribution to access the database. The parameters for the Zipf distribution are shown in the table and are the same used for the experiments in the last chapter.

The first four experiments examine the relative performance of $\mathcal{PT}$ and $\mathcal{PIX}$ for combinations of uniform or skewed client access patterns with flat or multi-level disks. For the flat

PrefetchOn	True
UpdateOn	False
BackChannelOn	False
NumClients	1
ThinkTime	2.0
ServerDBSize	3000
AccessRange	1000
CacheSize	1 (0.1%) to 1000(100%)
Δ	1,2,. . .4
θ	0.95
Offset	0, *CacheSize*
Noise	0%, 15%, 30%, 45%, 60%
Scatter	On, Off
RegionSize	50
$DiskSize_i$	<300, 1200, 1500>

Table 6.2: Prefetching Parameter Settings

disk, a 3000 page single-disk broadcast is used while for the multi-disk broadcast, a 3-disk program is used with the first (fastest) disk having 300 pages and the second and the third disk having 1200 pages and 1500 pages respectively. Analyzing the performance of $\mathcal{PT}$ under these varying conditions provides insight into how prefetching exploits client cache resources to reduce the expected delay for page requests.

The experiments in this chapter are based on the following assumptions which supplement those presented in Section 3.4:

- *Broadcast program is static.* The client population and their access patterns do not change. This implies that the content and the organization of the broadcast program remains static.
- *Read-only access.* There are no updates either by the clients or at the servers.
- *No backchannel.* Clients make no use of their upstream communications capability, i.e., they provide no feedback to servers.

6.3.2 Experiment. 1: Uniform Access, Flat Disk

In the first experiment, we examine the performance of the $\mathcal{PT}$ prefetching heuristic when the client access pattern is uniform (i.e., all pages in the *AccessRange* are accessed with equal probability), and the broadcast is a single, flat disk (i.e., $\Delta = 0$) and the *AccessRange* is *scattered* (as explained in Section 4.2.1). Figure 6.2 shows the expected delay for $\mathcal{PT}$ and $\mathcal{PIX}$ algorithms as the client cache size is varied from a single page to 1000 pages. In this case, $\mathcal{PT}$ and $\mathcal{PIX}$ have identical performance at either end of the spectrum, but $\mathcal{PT}$ has better performance throughout the entire intermediate range. When the cache size is 1 page, the client can

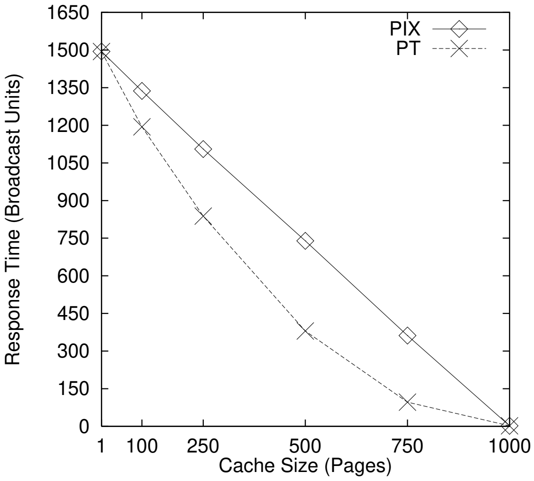


Figure 6.2: $\mathcal{PT}$ imitates Tag-team case
Uniform Access, Flat Disk

perform no caching or prefetching and thus, the expected delay is 1500 pages (this is one half of the time for one disk rotation) for both algorithms. As the cache size increases, the performance improves under both algorithms, until at a cache size of 1000 pages, the response time for both is zero (because all accessed pages are cached at this point).

Between the two endpoints, the prefetching performed by $\mathcal{PT}$ leads to better performance than $\mathcal{PIX}$. Note the case when the cache can hold exactly half of the accessed pages (Cache size = 500). At this point, the response time of $\mathcal{PT}$ is half that of $\mathcal{PIX}$. This performance improvement is exactly what is predicted by the Tag-team example described in Section 3.3.4, and occurs for similar reasons. In this case, $\mathcal{PT}$ and $\mathcal{PIX}$ have the same cache hit rates, but $\mathcal{PT}$ is able to reduce the cost of cache misses by better scheduling the cache residency of pages. To see why $\mathcal{PT}$ achieves the tag-team effect in this experiment, it is necessary to recall how the $\mathcal{PT}$ value of a given page changes over time (as described in Section 6.2). A page's $\mathcal{PT}$ value is at its maximum immediately after the page has been broadcast; it then linearly decreases until reaching zero at the instant the page is re-broadcast. As a result, using $\mathcal{PT}$, pages are more likely to be replaced from the cache as they get closer to being re-broadcast. In this experiment, which has uniform access and a flat disk, a cache size of half of the *AccessRange* allows $\mathcal{PT}$ to ensure that pages are cache resident for the first (most expensive) half of their inter-arrival gap, so that no cache misses occur during that half of the gap. Thus, as in Tag-team caching, the heuristic used by $\mathcal{PT}$ improves performance by reducing the penalty of cache misses.

In relative terms (i.e., % difference) $\mathcal{PT}$'s advantage over $\mathcal{PIX}$ grows with the cache size up to the point at which the entire access range is cached. However, the advantage in absolute terms (i.e., time) reaches its maximum at a cache size of 500, and decreases beyond that point

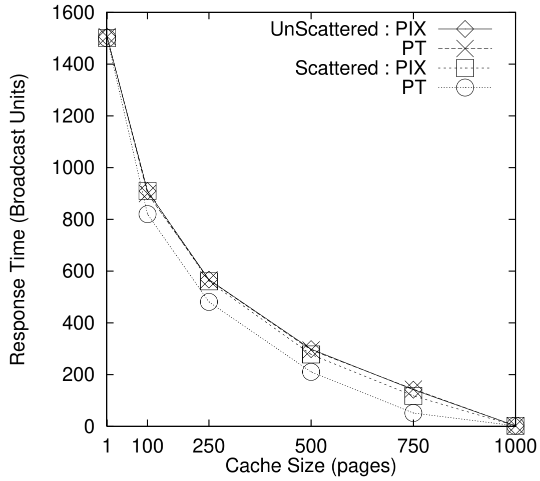


Figure 6.3: Effect of Scatter
Skewed Access, Flat Disk

in this experiment. The reason for this behavior is that every cache slot that is added to the cache adds 0.1% (i.e. 1 out of 1000 pages) to the client hit rate for both algorithms. The cost of a miss remains constant for $\mathcal{PIX}$ but decreases with the cache size for $\mathcal{PT}$, because the latter algorithm uses the additional cache slot to also fractionally shorten the delay for all pages. Thus, the marginal benefit of an additional cache slot is constant for $\mathcal{PIX}$ (each slot reduces the expected delay by 1.5 units), as can be seen by the linear decrease in response time for $\mathcal{PIX}$. In contrast, beyond the 500 page point, the marginal benefit of an additional cache slot decreases for $\mathcal{PT}$, so that the absolute benefit of $\mathcal{PT}$ over $\mathcal{PIX}$ decreases.

6.3.3 Experiment 2: Skewed Access, Flat Disk

The previous experiment examined the case when the client's access pattern is uniform and the disk is flat. This experiment instead studies the case when the client's access pattern is highly skewed (according to the Zipf distribution described in Section 4.1). The results of this experiment can be seen in Figure 6.3, which shows the response time for $\mathcal{PT}$ and $\mathcal{PIX}$ when used on two different flat disk configurations: *Scattered* and *Unscattered*. The Unscattered disk lays out all the pages in the *AccessRange* sequentially. When the Zipf distribution is used, this results in a layout where the accesses are highly clustered to a small region of the disk. This clustering is avoided in the Scattered disk by randomizing the location of the pages on the flat disk (as described in Section 4.2.1).

Turning to Figure 6.3, it can be seen that all four curves have similar performance. As in the previous case, performance improves with additional cache, but is non-linear due to the

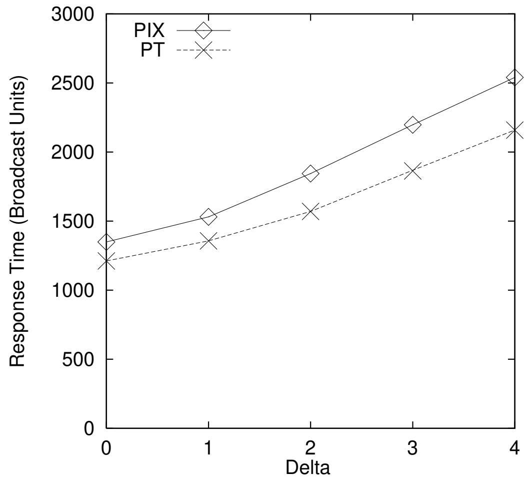


Figure 6.4: Uniform Access, 3-level Disk
Cache Size = 100, Varying Δ

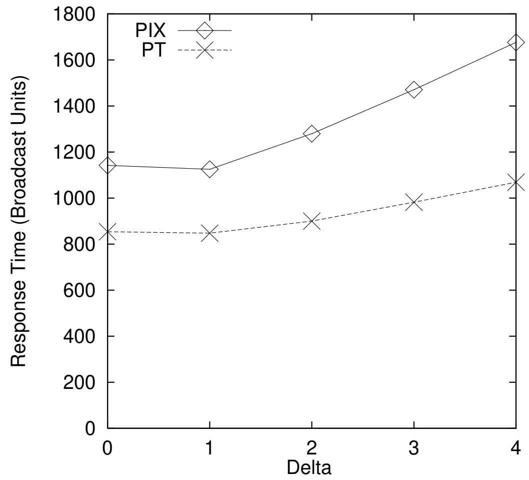


Figure 6.5: Uniform Access, 3-level Disk
Cache Size = 250, Varying Δ

skewed access pattern. As the cache becomes larger, the marginal performance gain from a new slot decreases here. When the Unscattered disk is used, $\mathcal{PT}$ provides no performance improvement over $\mathcal{PIX}$. With pages clustered together, $\mathcal{PT}$ cannot fully exploit the cache for tag-teaming pages with similar access probability, as the hot (i.e., high access probability) pages that appear earlier in the cluster tend to be pushed out of the cache by the hot pages that are ordered later in the cluster. The expected delay for the earlier pages, thus approaches half a rotation — as would be expected for a non-prefetching scheme. When the Scattered disk is used, $\mathcal{PT}$ has a noticeable performance improvement over $\mathcal{PIX}$ demonstrating the benefits of randomizing the (flat) disk contents for $\mathcal{PT}$-based prefetching. However, the performance advantage of $\mathcal{PT}$ over $\mathcal{PIX}$ is somewhat less than the previous experiment, as the skewed access pattern allows $\mathcal{PIX}$ to better exploit the client cache. Yet, for a cache size of 500, $\mathcal{PT}$ on the Scattered disk obtains a 20% improvement over $\mathcal{PIX}$.

6.3.4 Experiment 3: Uniform Access, Multi-Level Disk

The third experiment examines the behavior of $\mathcal{PT}$ when a multi-disk broadcast is used. For simplicity, a uniform client access pattern is used in this experiment. In this case, the broadcast consists of three disks, for which the fastest disk holds 300 pages, the medium disk holds 1200 pages and the slowest disk holds the remaining 1500 pages. The relative spinning speeds of the disks (and hence the arrival rates of pages on those disks) is varied using the disk skew parameter Δ. Figures 6.4 and 6.5 show the performance of $\mathcal{PT}$ in this case for client cache size is 100 and 250 pages respectively.

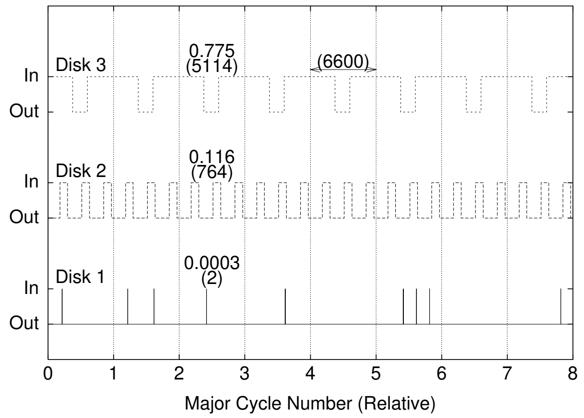


Figure 6.6: Cache Residency for $\mathcal{PT}$

Cache Size = 500, Δ = 2

In general, the two figures exhibit similar behavior. Due to the uniform client access pattern, increasing disk skew (Δ) harms performance here, as the best broadcast for a uniform access distribution is a flat broadcast. In both graphs, $\mathcal{PT}$ outperforms $\mathcal{PIX}$ and is less sensitive to Δ in the range of 0 (flat disk) to 4 (at which the pages on the fastest disk are broadcast nine times more frequently than pages on the slowest disk). The advantage that $\mathcal{PT}$ has over $\mathcal{PIX}$ is larger (in both absolute and relative terms) with a cache size of 250 pages than for a cache size of 100 pages. This is because $\mathcal{PT}$ can not fully achieve the Tag-team effect with a small cache. Likewise, if the cache becomes too large, then the absolute advantage of $\mathcal{PT}$ decreases. In this experiment, at a cache size of 250 pages (Figure 6.5), $\mathcal{PT}$ provides a substantial improvement over $\mathcal{PIX}$, and this advantage increases as the disk skew (Δ) is increased.

$\mathcal{PT}$ is more tolerant of disk skew than $\mathcal{PIX}$ in this case because it is effectively able to adjust the cache residencies of various pages in order to cope with their differing arrival rates. The manner in which $\mathcal{PT}$ accomplishes this can be seen in Figure 6.6, which shows the cache residency of a representative page in the AccessRange from each of the three disks of the broadcast. Since all pages in the AccessRange are accessed with equal probability, comparing the cache residency times of the three pages demonstrates the way in which $\mathcal{PT}$ exploits the client cache resources. The figure shown is for the case where the client cache size is 500 and Δ is set to 2. Time increases along the x-axis, which is divided into "major cycles" (i.e., periods of the entire broadcast). The three curves in the figure correspond to the pages from the three disks. When the curve is at the high position (denoted as "In"), the page is resident in the client's cache, and when the curve is in the low position (denoted as "Out"), the page is not in the cache. The figures on the graphs show the fraction of the cycle each page is cache-resident every time it enters the cache. The figure in parenthesis is the absolute time in broadcast units.

The top curve in the figure shows the cache residency of a page that is stored on the slowest

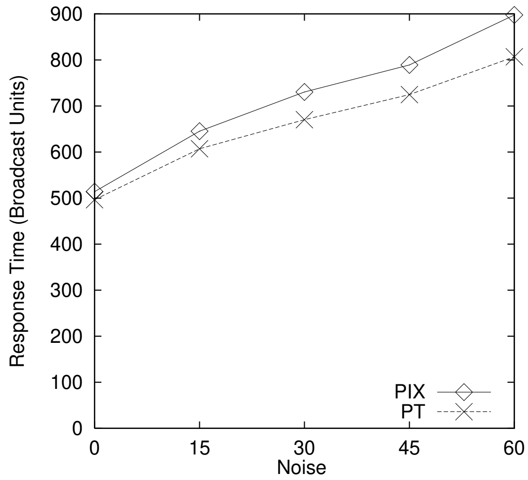


Figure 6.7: Skewed Access, 3-level Disk
CacheSize = 100, Δ = 2

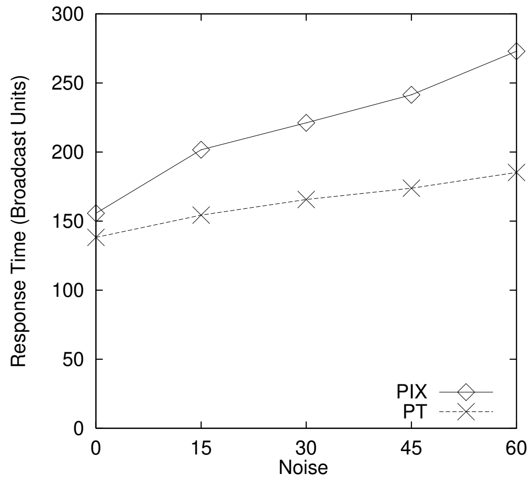


Figure 6.8: Skewed Access, 3-level Disk
CacheSize = 500, Δ = 2

disk (Disk 3). Pages on Disk 3 are broadcast once per major cycle, and therefore, $\mathcal{PT}$ removes the page from the cache once per major cycle. As the arrival time of the page nears, its `pt` value decreases, until the point it is replaced from the cache. The page is prefetched into the cache when it arrives again. The middle curve is for the page from the middle disk (Disk 2). Since Δ is set to 2, pages on this disk appear three times per major cycle, and therefore (as shown in the figure) $\mathcal{PT}$ expels the page from the cache three times during the cycle. Again, $\mathcal{PT}$ keeps the page in cache after it arrives, and expels it at the time closest to when it is due to arrive again. Comparing the curves for Disk 3 and Disk 2, it can be seen that $\mathcal{PT}$ gives more than twice the cache residency time to the page on the slower Disk 3 (77% per cycle) than to the page on Disk 2 ($0.116 \times 3 \approx 35\%$ per cycle).

$\mathcal{PT}$ is effectively determining the proper residency time for each page (based on its probability of access and its frequency of broadcast), and then distributing this time over the major cycle in a way that minimizes the expected delay for pages that have been removed from the cache. Finally, the bottom curve of Figure 6.6 shows the cache residency for a page that is stored on the fastest disk (Disk 1). Since this page is broadcast relatively frequently (5 times per major cycle), $\mathcal{PT}$ does not allocate any cache residency time to it. As a result, the page is brought into the cache only when there is an outstanding client request for it, and is replaced from the cache immediately. This is reflected by the fact that it stays only for 2 broadcast units (0.0003% of the cycle) in the cache – the think time before the next access. This behavior can be observed as the random spikes on the lowest curve in the figure.

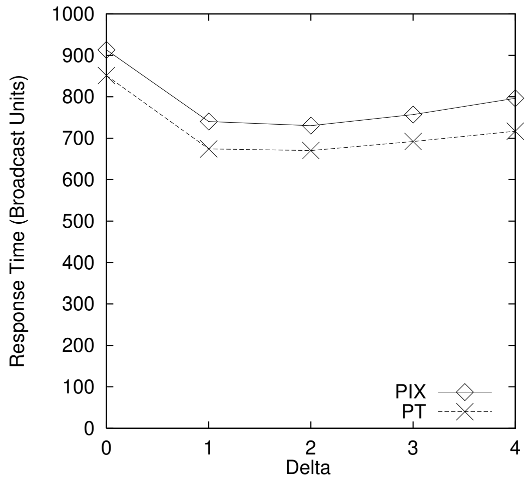


Figure 6.9: Skewed Access, 3-level Disk
CacheSize = 100, *Noise* = 30%

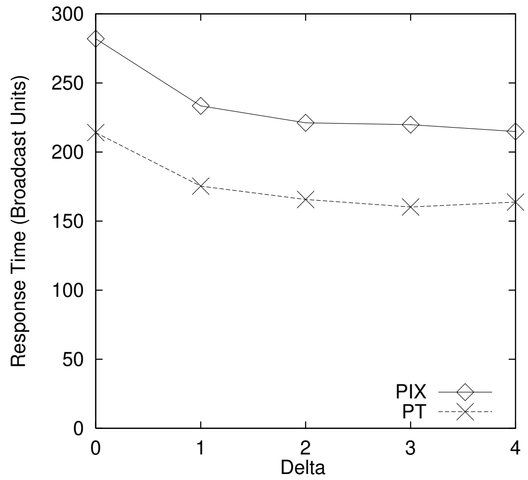


Figure 6.10: Skewed Access, 3-level Disk
CacheSize = 500, *Noise* = 30%

6.3.5 Experiment 4: Skewed Access, Multi-Level Disk

The fourth experiment investigates the combination of a skewed Zipf client access pattern (as was used in Experiment 2) with the multi-disk broadcast. Figures 6.7 and 6.8 show the performance of $\mathcal{PT}$ and $\mathcal{PIX}$ as *Noise* is varied using the 3-disk broadcast of the previous experiment with Δ set to 2, for the cache sizes of 100 pages and 500 pages respectively. Recall that increasing the noise parameter increases the discrepancy between the client access probabilities and the page broadcast frequencies chosen by the server. A higher degree of noise models the effect of a larger client population with differing access profiles.

When the cache is small (Figure 6.7), the performance of both algorithms is similar, and both are negatively impacted by increasing noise. With the larger cache (Figure 6.8), the performance of $\mathcal{PT}$ is much more robust. $\mathcal{PT}$'s tolerance of noise is due to its ability to fine tune the cache residencies of pages to account for their broadcast frequencies (Section 6.3.4). When the server places an important page on the slow disk, $\mathcal{PT}$ is able to compensate by giving that page more time in the cache, and ensuring that the time when the page is not cache resident is placed as close as possible to the subsequent arrival of the page on the broadcast.

Figures 6.9 and 6.10 show data points from the same experiment, with noise fixed at 30%, and Δ varied from 0 (i.e., a flat disk) to 4. In this case, it can be seen that with a large cache, $\mathcal{PT}$ is able to provide a significant performance improvement over $\mathcal{PIX}$ for a wide range of different disk shapes. Thus, it can be seen that $\mathcal{PT}$ is a very robust metric — it can better exploit client cache resources across a range of client access distributions, and broadcast programs, and can also help mitigate the negative performance impact of disagreement between client access probabilities and server broadcast frequencies (as represented by "noise").

6.4 $\mathcal{APT}$: Approximating $\mathcal{PT}$

The previous sections have shown the potential benefits of prefetching in a broadcast environment. However, the prefetching heuristic $\mathcal{PT}$ in its pure form is impractical— it requires $O(CacheSize)$ operations at every "tick" of the broadcast to determine the page with the lowest `pt` value. As a result, based on the insights gained in studying $\mathcal{PT}$, this section presents the design and performance numbers of an implementable approximation of $\mathcal{PT}$, called $\mathcal{APT}$.

$\mathcal{APT}$ takes the cached pages, sorts them based on probability of access and partitions this ordering into contiguous, non-overlapping regions. All pages in a region are assumed to have the same probability of access. Thus, each region has only one potential candidate for a replacement victim — the page in the region which arrives next on the broadcast, (i.e., the one with the smallest `t` value). $\mathcal{APT}$ evaluates `pt` values for each of these potential victims and evicts the one with the lowest `pt`. Consequently, the eviction decision for $\mathcal{APT}$ has a complexity of $O(Number\ of\ Regions)$ as opposed to $O(CacheSize)$ for $\mathcal{PT}$. Two remaining implementation problems must be addressed: 1) maintaining in constant time the next broadcast page in each region, and 2) determining the access probabilities of the client. We next address these problems in order.

Consider Figure 6.11. It shows an $\mathcal{APT}$ implementation with three probability regions for a flat (single-disk) broadcast. The pages in a region are organized in a doubly-linked circular chain sorted by their order of arrival in the broadcast. Region 1 shows examples of pointers connecting neighboring slots. Each region also has a pointer to that region's potential replacement victim (*PRV*), (i.e., the page with the smallest `t`). In the figure these are shown as shaded slots (pages g, n and q). Thus, one of the *PRV* must be the page with the smallest `pt` value since all pages in a region are assumed to have the same access probability. When a *PRV* is evicted (say g), the page following the old *PRV* becomes the new *PRV* (in this case, page h). Let the client access a new page z and q be chosen as the victim. Region 3 is then updated by freeing the slot for q and making r the new *PRV*. The new page is assigned a region based on its current estimated probability of access. Let z be allocated to Region 2. It is then inserted into the second chain before the *PRV* (n in the figure) because the *PRV* has not yet been broadcast and the new page will have the largest value for `t`. The size of the chains change dynamically as can be seen for regions 2 and 3 in the figure.

The implementation described above is for a flat broadcast; however, maintaining such regions for each disk easily extends this technique to handle a multi-disk. The problem in that scenario generalizes to finding the *PRV* for each region for each disk. This extension requires no additional space and only a small constant number of extra pointer operations proportional to the number of disks.

As the number of regions (*NumRegions*) increases, the closer the caching decisions will approach those of $\mathcal{PT}$. Making *NumRegions* too large, however, will have an adverse effect on

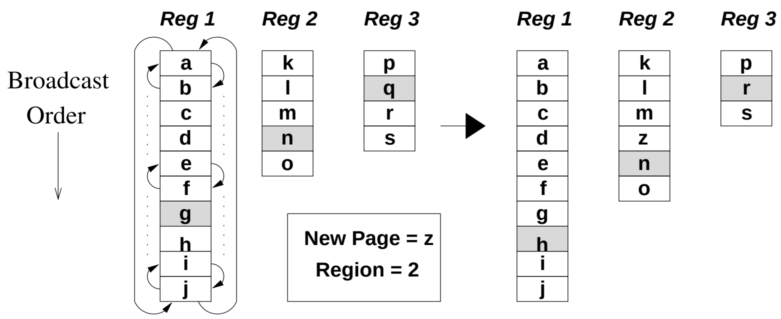


Figure 6.11: Page Replacement in $\mathcal{APT}$

the performance of $\mathcal{APT}$ itself since its complexity is proportional to the number of regions. Our studies have shown that four regions give acceptable performance for the distribution under consideration. In the experiments that follow, we determined the probability range of each region by dividing the difference of the highest and the lowest access probability of any page in the database into *NumRegions* equal sized ranges. For example, let 0.001 and 0.041 be the lowest and the highest probability values and let the number of regions be four. Then the ranges of the regions are [0,0.011), [0.011,0.021), [0.021,0.031) and [0.031,1). Note that the first and the last regions have been extended to 0 and 1 to account for any pages which may, at some point, fall outside the range. A page's current probability determines the region it gets assigned to. This simple approach gave us acceptable performance.

Unlike demand-driven caching, prefetching also requires the access probabilities of pages not in the cache to determine if they are worth prefetching. We use a technique similar to the 2Q approach proposed in [JS94] of maintaining a second queue (twoQ). When a page is evicted from the cache, its page number and its probability of access is entered into the twoQ in a FIFO fashion. A page is considered to be a prefetch candidate only if it is in the twoQ. For the moment, we assume $\mathcal{LIX}$ has a probability estimate for each page. In Section 6.4.2, we describe the model used in the following experiments to generate these numbers.

6.4.1 Experiment 5: $\mathcal{LIX}$ vs. $\mathcal{APT}$

In this section, we compare the performance of $\mathcal{APT}$ with $\mathcal{LIX}$. $\mathcal{LIX}$ was introduced in Chapter 5 as an efficient constant time approximation of $\mathcal{PIX}$. $\mathcal{LIX}$ maintains an LRU-style chain ordered by probability of access for each disk of a multi-disk broadcast. $\mathcal{LIX}$ also keeps a running probability estimate for each broadcast page based on its past history of access.

$\mathcal{LIX}$ uses a very simple formula to estimate the probability of a page. To be fair, in the following experiments $\mathcal{LIX}$ was also run using the same probability model as was used by $\mathcal{APT}$. We call this algorithm $\mathcal{LIX}(2)$.

Consider Figure 6.12. It shows the same simulation space as Figure 6.9, i.e., *CacheSize* =

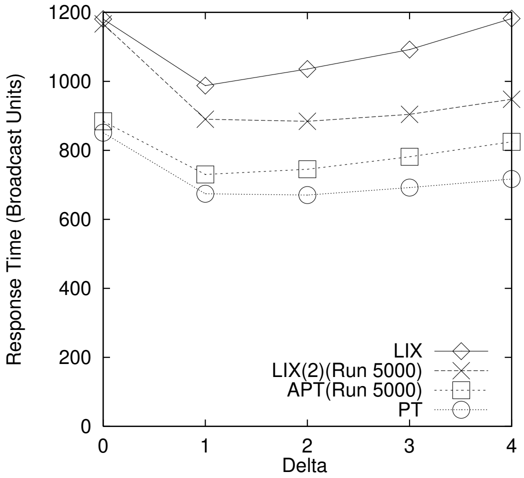


Figure 6.12: $\mathcal{APT}$ Performance, Cache=100

Skewed Access, Varied Δ, *Noise* = 30%

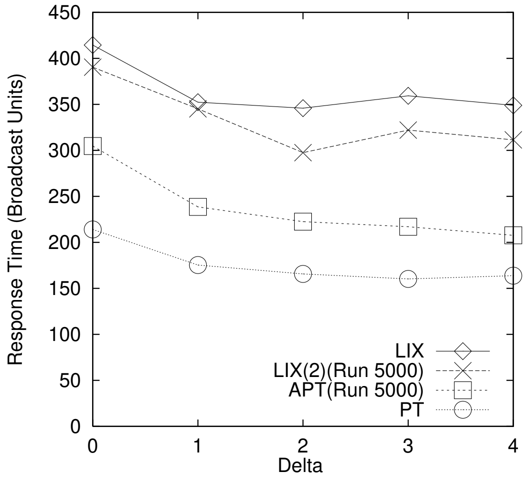


Figure 6.13: $\mathcal{APT}$ Performance, Cache=500

Skewed Access, Varied Δ, *Noise* = 30%

OffSet = 100 and a *Noise* of 30%. The size of twoQ in this experiment was twice the cache size, i.e, 200 pages. In this experiment, $\mathcal{LIX}$ performs the worst and quickly degrades as Δ increases beyond 1. $\mathcal{LIX}$(2) does significantly better than $\mathcal{LIX}$ reflecting the difference between the two probability estimation techniques. The last two lines show the performance of $\mathcal{APT}$ and $\mathcal{PT}$. The ideal algorithm $\mathcal{PT}$ outperforms $\mathcal{APT}$ by a small margin. $\mathcal{APT}$ improves on the response time of $\mathcal{LIX}$(2) by 10-30%. In these experiments *NumRegions* was set at 4.

Figure 6.13 shows the performance of the four algorithms for a cache size of 500 for the same simulation space. The size of twoQ here is equal to the cache size. The absolute response time for all the algorithms is lower because of the larger cache. The performance improvement of $\mathcal{LIX}$(2) over $\mathcal{LIX}$ is not as significant here as in the previous case. $\mathcal{APT}$ follows $\mathcal{PT}$ throughout the space shown but not as closely as in the last experiment. This is a consequence of the probability model used by $\mathcal{APT}$ (and $\mathcal{LIX}$(2)). The probability model (which is described in the next section) produces poorer approximations for pages with low (real) probability of access. Since the pages which are not cache-resident are those with a low probability of access, $\mathcal{APT}$ (and $\mathcal{LIX}$(2)) tend to make more errors. In spite of this handicap, $\mathcal{APT}$ provides about a 25% performance improvement $\mathcal{LIX}$(2) and is more robust over a wider spectrum of disk configurations (as determined by Δ).

So far, it has been assumed that the client has a fairly good knowledge of its access probabilities. This might be too strong an assumption. In the next section, a probability model is developed and the sensitivity of the model is investigated on how accurate these probabilities must be in order to retain the acceptable performance.[1]

[1]The results presented in this section for $\mathcal{APT}$ and $\mathcal{LIX}$(2) were for a mean error in the probabilities of about 0.4. The precise definition of error will be given in the next section.

6.4.2 Experiment 6: Sensitivity of $\mathcal{APT}$

As we observed earlier, the $\mathcal{APT}$ algorithm relies on the client's knowing its own access probabilities. We assume that the client builds a model of these probabilities by sampling its own requests over some period of time. We count the number of accesses for a page and divide this number by the total number of accesses to compute the page's probability. As long as the access pattern is not changing, a longer sampling period should produce more accurate results. Such counting-based schemes have been proposed before for estimating probabilities for page replacement. The CLOCK [SPG91] and the frequency-based cache management algorithm introduced in [RD90] are two examples.

In general, such a probability model will likely not be completely accurate. This section describes experiments which analyze $\mathcal{APT}$'s sensitivity to inaccuracies in the probability estimation. Different levels of accuracy are generated by varying the sample length in the learning run.

We use probability distributions that are derived from an initial run whose length (*InitRun*) ranges from 500 to 50,000 accesses. Let p_i and P_i be the approximate learned probability and the real probability of page i respectively. The error e_i for page i is given by the formula:

$$e_i = (|p_i - P_i|)/P_i$$

Thus, when $e_i = 1$, $0 \leq p_i \leq 2P_i$. The cumulative error in the distribution is expressed by the mean and the variance for these individual error values over all pages. The cumulative errors for the learned distributions that we used in the experiments are summarized in Table 6.3.

Figure 6.14 shows the response time of $\mathcal{APT}$ for different lengths of the learning run (*InitRun*). The simulation space is the same as Figure 6.12, i.e., *CacheSize=OffSet=100*, *Noise* = 30%. In general, higher sampling rates produce better performance. For the shorter initial runs of 500 and 1000 accesses, the response time is somewhat higher than the rest. However, beyond 5000 accesses, the performance does not improve substantially. The results in Section 6.4.1 were for sample length of 5000. From Table 6.3, the error rate for this length of run is 0.437 with a variance of 0.146. This means that the approximate probability can be anywhere from

Learning Run Length	Mean Error	Variance
500	1.380042	1.764039
1000	1.062268	0.739920
2000	0.705625	0.319217
5000	0.437711	0.146137
20000	0.227719	0.035724
35000	0.176085	0.022073
50000	0.146691	0.016047

Table 6.3: Approximation Error vs. Length of Run

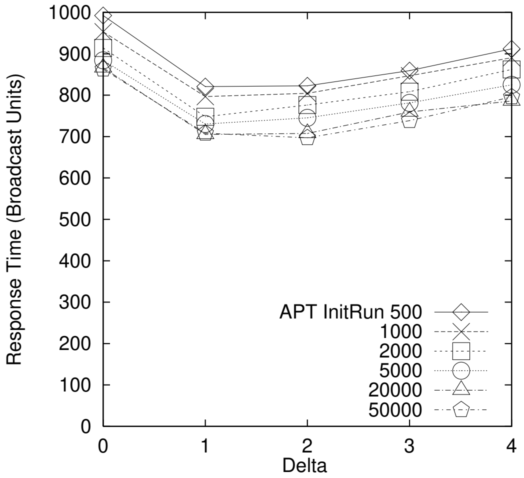


Figure 6.14: Learning Sample Sensitivity
Cache = 100, Varied Δ, Noise = 30%

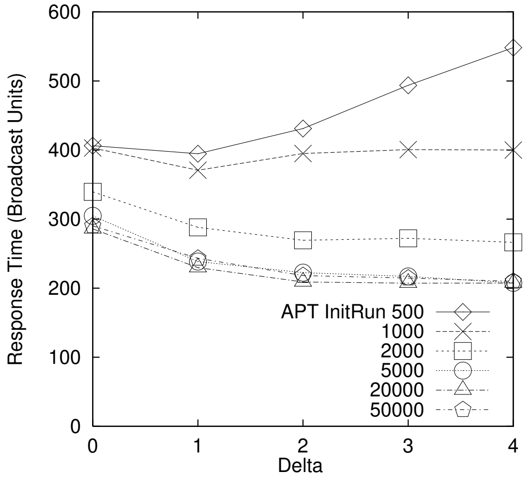


Figure 6.15: Learning Sample Sensitivity
Cache = 500, Varied Δ, Noise = 30%

half to 1.5 times the real probability. In spite of this error, $\mathcal{APT}$ does significantly better than $\mathcal{LIX}$. Figure 6.15 shows the same space for a cache size of 500. While the absolute numbers are different as expected, like for the small cache, beyond a learning run of 5000 accesses, the performance remains the same.

We conclude from this that $\mathcal{APT}$ is fairly insensitive to perfect knowledge of the access distribution. This is the case since $\mathcal{APT}$ categorizes pages based on probabilities and in our distribution, the hot pages are very hot and the cold pages are quite cold. Thus, it is unlikely that pages will be incorrectly categorized. This is acceptable since multi-disk broadcasts are beneficial only in environments where the access distributions are skewed.

Figure 6.16 shows the performance of $\mathcal{APT}$ with 2, 4 and 6 regions with an initial learning run of 5000 accesses. The simulation space is the same as in Figure 6.15 and the learning run length is 5000 accesses. The performance is worst for two regions as would be expected. Increasing the number of regions from four to six does not produce any significant difference for $\Delta \geq 1$. In fact, for higher Δ, $\mathcal{APT}$ with 6 regions actually does slightly worse. This is because, given the approximate knowledge of probabilities, trying to create too many regions may cause pages to get wrongly classified. As the accuracy of the probability estimation increases this effect is no longer seen. This reflects a tradeoff between the error in probability estimation and the number of regions. As the noise increases, the lines spread apart with 6 regions giving the best performance and 2 regions the worst. However, increasing the number of regions, inflates the cost of the algorithm. On an average, for the simulation space studied, four regions seem to give the best performance. These experiments show the $\mathcal{APT}$ is insensitive to probability estimation errors and is an efficient and robust approximation of $\mathcal{PT}$.

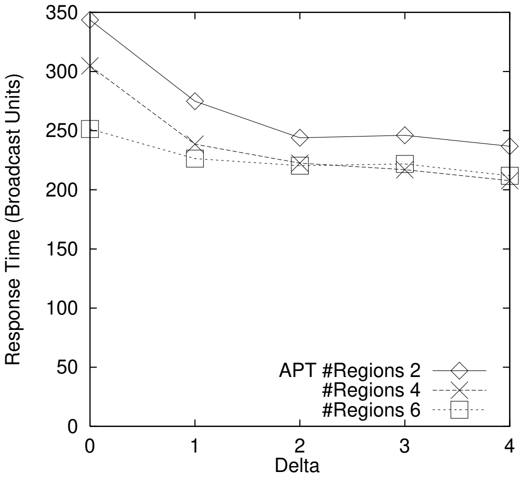


Figure 6.16: $\mathcal{APT}$ Response for different #Regions

Cache = 500, Varied Δ, Noise = 0.30, *InitRun*=5000

6.5 Related Work

Prefetching has been studied extensively as a technique to improve user performance in various areas including databases, operating systems and processor architectures. Prefetching techniques usually fall into two categories – prefetching based on user hints and prefetching based on past access history.

[PG94, PGG+95] propose Transparent-Informed Prefetching which exploits I/O concurrency in disk arrays using hints from applications. The other class of prefetching algorithms is based on inferring access patterns from a stream user requests ([KE91, PZ91, CKV93, GA94, KTP+96] to name a few). Each of these techniques uses the access stream to develop a model of the user behavior. The approach taken in this chapter for prefetching in the broadcast environment leans towards the second category in the sense that we develop a model of client behavior based on past accesses. However, performance improvement in a broadcast environment comes not from achieving a higher hit rate but from reducing the cost of a cache miss. Consequently, the crux of our prefetching heuristic is not developing a better access model (which is sufficient to produce a better hit rate) but rather, using that knowledge to efficiently evaluate the cost metric like the `pt` value. Our simple counting-based probability estimator can, thus, be replaced by any of the other more sophisticated models to produce even better results.

There has been work on prefetching in Teletext systems [AW87, Won88]. In [AW87], a caching strategy called the Linked Page scheme is described which uses embedded links in each page to decide what to prefetch next. In that technique, these links are sorted by the probability of next access. Based on the cache space available, pages are prefetched in a straightforward manner in a decreasing order of probability. Stashing and Hoarding used in the mobile computing environment [KS92, KPR94, TLAC95] are very similar to prefetching in that pages are pre-cached before they are actually accessed. However, their focus is to improve availability in the file system when the client is *disconnected* from the server as opposed to improving performance which is the goal in our study.

6.6 Conclusions

In this chapter, the notion of prefetching was examined in the context of the Broadcast Disks framework. Compared to demand-driven caching introduced in Chapter 5, prefetching is a more opportunistic use of the cache resources where pages are brought into the cache prior to the actual access. The simple example of tag-team caching (Section 3.3.4) demonstrated how prefetching opens up newer opportunities to utilize cache slots and enhance responsiveness.

We introduced a prefetching heuristic $\mathcal{PT}$, which outperformed $\mathcal{PIX}$, the demand-driven algorithm from last chapter, in all the cases studied. $\mathcal{PT}$ uses the time until next arrival as a metric for the cost of re-acquisition. Unlike $\mathcal{PIX}$, which uses the *average* cost of re-acquisition in the replacement decision, the time metric gives $\mathcal{PT}$ an estimate of the *instantaneous* cost. By using the instantaneous cost, $\mathcal{PT}$ is able to generate free cache slots by dropping pages in those portions of the broadcast cycle when their instantaneous cost is much lower than their average cost and prefetching the pages back again when they are re-broadcast (when the instantaneous cost is higher). This cycling of pages in and out of the cache allows $\mathcal{PT}$ to significantly improve the effective size of the cache real-estate and that is the cause of its superior performance.

Finally, the design and implementation of a constant time approximation to $\mathcal{PT}$, called $\mathcal{APT}$ was described. $\mathcal{APT}$ is based on LRU and uses a simple estimation scheme to generate probability values for various pages. Experiments using $\mathcal{APT}$ show it is not very sensitive to the errors in estimation and is a good approximation to $\mathcal{PT}$ even when the probabilities are significantly different from the real values.

The techniques described in this chapter and the experimental validation demonstrate that prefetching truly exploits the characteristics of the Broadcast Disks model and is inexpensive to deploy in practice. Since prefetching in this environment is much less risky than in traditional systems, it has tremendous potential as a basis for cache management in dissemination-based systems.

An implicit assumption made in this study is that the client is able to sample every page for prefetch and bring it into the cache if necessary. However, in reality this may not always be possible. For example, the page may be corrupted due to error on the channel or, if network has extremely high bandwidth, the client may not be able to keep up with the broadcast and start missing pages. In such scenarios, prefetching heuristics like $\mathcal{PT}$, have a potential for poor performance. This is because when $\mathcal{PT}$ evicts a page from the cache, it typically expects to pick it up when it is re-broadcast again. Failing to do so, for the aforementioned reasons, can upset its metric calculations, leading to poor cache utilization. This is a critical issue for prefetching algorithms and we will investigate it further using the prototype in Chapter 9.

The inability to accurately predict the arrival of a page (due to page drops) also has an impact on the design of an approximation for $\mathcal{PT}$. In order to reduce the O(*CacheSize*) complexity of $\mathcal{PT}$ to a constant number of operations, $\mathcal{APT}$ assumes that groups of pages have the same probability of access. In other words, $\mathcal{APT}$ approximates the `p` part of the `pt` metric while using the exact values of `t`. An alternative approach to approximating $\mathcal{PT}$ would be to approximate `t`. This would imply that groups of pages which appear close to each other on the broadcast would be assumed to have the same time of arrival and the problem would then reduce to finding the page in each group which has the lowest probability estimate. This approach would have two advantages. One, the probability model would not need to be built *a priori* (as in the $\mathcal{APT}$ implementation presented), and thus, the algorithm would be more reactive to access pattern changes. Secondly, by using a fuzzy value for the time to next arrival, the implementation will be more immune to dropping pages unlike $\mathcal{APT}$, which has the potential for performance degradation (as in $\mathcal{PT}$) when the client fails to keep up with the broadcast. Developing such an approximation with a constant time complexity is an interesting avenue for future work.

Chapter 7

Disseminating Updates

7.1 Introduction

The focus of the experiments of the previous chapters was on the performance of the Broadcast Disks model in the context of a read-only environment, where neither the broadcast program nor its contents were changed. These studies uncovered fundamental properties of the approach and demonstrated the impact of push-based data delivery on client cache management. Clearly, however, many of the applications that can best profit from a broadcast-based approach also contain inherently volatile data. Examples of such applications include: stock quotation systems, weather and traffic reports, and logistics applications. Thus, for any push-based system to be successfully deployed in practice, it is imperative for updates to be disseminated efficiently.

Updates in any client-server environment introduce a very key performance–correctness tradeoff. In such systems, clients access data from their local caches, while updates to data values (either initiated at the server or by one of the clients) are collected at a server site. In order to keep the clients' caches consistent with the updated data values, the copies of modified items in the client cache must be invalidated or updated by the server. Different applications, however, have differing consistency demands. Some applications need to access only the most recent values of data items, while others can tolerate some degree of staleness. Furthermore, in some consistency models, such as ACID transactions, consistency must be preserved across reads of multiple data items. In order to provide correctness, the environment has to provide a consistency guarantee no weaker than the requirement of the client application. Typically, however, the stronger the consistency guarantee, the greater the processing and communication overhead which can hurt client performance. Every environment dealing with volatile data has to balance this tradeoff between performance and correctness.

Broadcast Disks and other push-based environments face exactly the same tradeoff. Since the clients may not have access to a backchannel to connect to the server, in these environments the burden of keeping the client caches consistent is the sole responsibility of the server. However, the dissemination nature of the environment and the periodicity of the broadcast makes it more conducive for such applications than a traditional pull-based environment. In fact, the Broadcast Disks model has at least three advantages over a traditional client-server system:[1]

- *Server-initiated Notification*. In the applications being considered, the updates are collected at the central server. In a traditional environment, clients would potentially have to poll the server continually to know when the values have changed. In a push-based scheme, the server need inform the clients only if (and when) the data item is updated, thereby significantly cutting down the communication the overhead.[2]

- *Single Notification*. If the data delivery is via a broadcast channel as in the Broadcast Disks environment, a single notification suffices in informing all the interested clients about the update. In a pull-based system, each client would have to independently discover this information.

- *Periodic Refresh*. The periodicity of the Broadcast Disks model ensures that the locally cached values at the client are automatically refreshed at least once a broadcast cycle (or more, if a multi-disk schedule is used).

The dissemination-nature of the broadcast environment challenges some of the basic assumptions made in traditional systems for dealing with updates. In this chapter, we will re-evaluate this performance-correctness tradeoff in the broadcast setting and investigate how the standard techniques to support updates can be adapted to this framework. Broadly, this chapter will address the following three issues:

- *Validate the read-only intuitions*. The results presented in the previous chapters were based on a read-only model of the database. In this chapter, we will try to verify if the intuitions and techniques developed in the read-only world scale into an update setting.

- *Consistency Models*. There have been a number of consistency models proposed in the literature. In the next section, we will highlight some of these and study which of them are appropriate in the broadcast environment. This chapter will also investigate, the role listening to the broadcast continually (or, being intermittently connected) plays in determining the consistency model that can be supported.

[1]The assumptions made in Section 3.4 are still valid here including the assumption that clients are continually sampling the broadcast program.

[2]Some client-server database and file systems already use some form of server-initiated notification schemes ("callbacks") [How88, NWO88, Fra96]. However, they require the server to maintain state information about the clients who have cached copies. These schemes can be expensive and they have to address recovery in case of server failure.

- *Performance and scalability in presence of updates*. As pointed out in the earlier chapters, the Broadcast Disks environment is attractive because of its scalability. This chapter investigates the extensions needed to maintain this scalability advantage and client performance in the presence of updates.

The following assumptions hold about the broadcast environment in the experiments that follow. These supplement the assumptions already listed in Section 3.4.

1. *Broadcast program is static.* The structure of the broadcast program does not change. Updates merely replace the value of the data that is broadcast in each page. This corresponds to the *Data Values* change for dynamic broadcast programs (Section 3.2.3).

2. *All updates are performed at the server.* As stated previously, we assume a system in which clients access data from the broadcast in a read-only manner, while updates are first posted at the server and subsequently disseminated.

3. *Clients have no back-channel capacity.* In the experiments that follow, clients do not have a mechanism for sending messages to the server. Thus, for example, clients can not send special requests to the server for missing or invalidated data items.

4. *Active clients continuously monitor the broadcast.* We assume an environment in which active clients can constantly monitor the broadcast. If a client stops monitoring the broadcast, then it can no longer trust the contents of its cache. Techniques for implementing cache consistency for intermittently connected clients have been studied in [BI94, JBEA95, WYC96].

5. *Clients remain active for significant periods of time.* Once a client begins monitoring the broadcast, it does so for a period of time sufficient for its cache to fill and reach a steady state. We, thus, focus on the steady state behavior of connected clients.

The outline of the rest of the chapter is as follows. The next section proposes several models of consistency relevant to a dissemination-based environment. Section 7.3 introduces the basic techniques used to disseminate updates in this environment. Then, the simulation experiments in the update scenario are described. Finally, Section 7.6 summarizes the results of the update study.

7.2 Models of Consistency

As stated in the introduction, allowing disseminated (and possibly cached) data values to be updated creates the potential for data consistency problems. The notion of data consistency is, of course, application-dependent. In database systems, data consistency is traditionally tied to the notion of transaction serializability. In practice, however, few applications demand or even want full serializability, and much effort has gone into defining weaker forms of correctness (e.g., [BBG+95, Kor95]). Because dissemination-based information systems are only now beginning to emerge, the notion of data consistency for applications in such systems is even less well-understood.

This chapter focuses on a Broadcast Disks environment where all updates are collected at the server and clients access the data in a read-only fashion. Such an access pattern is likely in many data dissemination applications such as stock quotation systems, news and weather services, etc. In such an environment there are a number of reasonable choices for consistency models. These include:

- *Latest Value* - Clients must always access the most recent value of a data item. This level of consistency is what would arise naturally if clients performed no caching and servers broadcast only the most recent values of items. Note that this model, although often used in dissemination-based and/or mobile database work, is far weaker than serializability — there is no notion of mutual consistency among groups of items.

- *Quasi-caching* [ABGM90] - Consistency can be defined on a per-client basis using constraints that specify a tolerance of slack compared to *Latest Value*. Some examples of such constraints include values that must be within x% of the latest value (e.g., for scalar attributes) or within y minutes of the latest value or within the last z changes of the value. Quasi-caching gives clients the ability to use cached data items in cases where they may otherwise be unsure of their correctness, and may allow the server to more lazily disseminate updates when the caching constraints of the clients are known.

- *Periodic* - Data values only change at specified intervals. In a Broadcast Disk environment, such intervals can be tied to the major or minor cycles of the broadcast. If clients are allowed to cache data, then they are guaranteed that such data will remain valid for the remainder of the interval during which the data was initially read. Such an approach was used in the Datacycle system [BGH+92].

- *Serializability* - If client reads and server updates are performed within the context of transactions, then serializability (or some reduced degrees of isolation [BBG+95]) may be an appropriate notion of consistency. Datacycle implemented serializability using a combination of periodic update, optimistic concurrency control at clients, and the broadcasting of update logs.

- *Opportunistic* - For some applications (or under certain conditions) it may be acceptable to read any version of a data item that is available. This notion of consistency is implemented by allowing clients to freely access data from their cache without worrying about consistency. Such an approach has obvious advantages for applications where long-term disconnection from the server is likely. In some cases, reading data of questionable quality may be preferable to having no data access at all.

In this chapter, we focus on two models of consistency: *Latest Value* and *Periodic*. These models allow us to study the consistency maintenance techniques under two different levels of overhead in a manner that is relatively independent of application semantics. *Latest Value* is likely to cause the greatest overhead for update dissemination— every update may require data cached at clients to be immediately invalidated or updated. Also, this model forces the clients to monitor the broadcast continually. Thus, *Latest Value* provides a good test of the efficiency of the mechanisms used to ensure consistency. The *Periodic* model, which places fewer demands on the efficiency of consistency maintenance, is useful for several reasons. First, it fits well with the cyclic behavior of the Broadcast Disk model. Also, it is similar to the basic approach used in Datacycle, and, as shown in the Datacycle work, it can be used as a foundation on which to build a serializable model. An advantage of the periodic model is that it does not require the client to listen to the broadcast all the time to maintain correctness. Since the data are scheduled to change only at certain intervals (at the beginning or the end of a period), it suffices for the client to monitor the broadcast during that time to determine which pages are out of date. In this chapter, we study the periodic model in which the consistency actions are performed once per major cycle.

Compared to the *Latest Value* and *Periodic* models, the *Quasi-caching* and *Opportunistic* models depend more heavily on the semantics of data access in specific applications. The *Serializable* model is relevant only in the presence of transactions. Since in this thesis, the clients access data on a per-item basis and are not transactional, this model is not being considered. The focus of the experiments that follow is on the fundamental tradeoffs involved with the mechanisms for maintaining data consistency in the Broadcast Disks environment; thus, we perform our experiments using the *Latest Value* and *Periodic* models. Studies of the other three models, however, are an interesting avenue for future work, particularly in the context of specific applications.

7.3 Consistency Techniques

The implementation of data consistency requires mechanisms to ensure that clients see only valid states of the data, or at least, do not unknowingly access data that is *stale* according to the rules of the consistency model. Each model of data consistency places demands on both the server broadcast and on client cache management. The server is responsible for informing clients either prior to or at the instant that a page is broadcast that their cached copy may be stale. In the broadcast environment, where servers have no knowledge of the contents of client caches, the server must broadcast consistency information for any clients that may be affected. On the other hand, it is the responsibility of the clients to obtain the consistency information and take appropriate actions. In the case where a client may have missed consistency information due to, say, disconnection or to broadcast errors, the client must avoid accessing any mutable cached data until it can ascertain the validity of its cache contents.

Problems related to consistency arise in most other systems where client caching is used, such as multi-processor architectures [AB86], distributed file systems [LS90], distributed shared memory [NL91], and client-server database systems [Fra96]. In such systems, there are two basic techniques for communicating consistency information: invalidation (also called Write-Invalidate) and propagation (also called Write-Broadcast). For invalidation, the server sends out messages to inform the client of modified pages. The client removes these pages from its cache. For propagation, the server sends updated values, and the client replaces its old copy with the new one. The tradeoffs between these two approaches in traditional pull-based client-server databases is examined in [FC92, Fra96].

Invalidation and propagation are also useful techniques in the Broadcast Disk environment. The nature of this new medium, however, changes the relative tradeoffs between them and introduces some additional opportunities. In this section, we briefly describe how these techniques can be adapted to the broadcast paradigm.

7.3.1 Invalidation

The quickest way to ensure that clients do not access stale data is through the use of invalidation. Invalidation messages can be quite small, requiring only that the page that has become invalid be identified. When a client receives an invalidation, it checks to see if it has a locally cached copy of the affected page, and if so, removes the page from its cache (or simply marks it "invalid"). In this paper invalidations are implemented through the use of an *invalidation list*. An invalidation list is simply a structure that contains the identifiers of pages that should be invalidated at a particular instant. Under the Periodic consistency model, updates are saved up and an invalidation list containing possibly multiple page identifiers is broadcast at a pre-specified interval, such as at the beginning of a major or minor cycle. In the experiments studied in this chapter, invalidation lists contain the identifiers of all pages that have been updated since the last invalidation list was broadcast.

Invalidation, while efficient to implement, can have a dramatic effect on the performance of client caches. When a client first "tunes in" to a broadcast program, it faults in (or prefetches) pages from the broadcast until its cache is full. From that point on, the cache is managed using a *cache replacement policy* — if a new page is to be brought into a full cache, an existing cached page must be chosen as a victim and replaced from the cache. The phase in which a client is initially loading its cache is known as the *warm-up phase*. Once the cache has been full for some time, and data is being cycled in and out of the cache, the client is said to be in *steady state*. In the Broadcast Disk model, the best broadcast program for clients in the warm-up phase is different than the best program for clients in steady state. As described in Section 5.3, during steady state the best broadcast is a program with an *Offset*.

Invalidations, however, have the effect of moving client caches away from steady state. If a page is updated, it is removed from the client's cache regardless of how recently or how often it has been accessed. Recall that the role of the cache in this environment is to mitigate the effect of poorly designed broadcast programs (from the client's perspective) by selectively caching pages whose local access probability does not match the server's global probability. However, invalidations upset this balancing act done by the client by removing pages independently of the cache management strategy. Thus, invalidations undermine the steady-state assumption on which the use of *offset* is based. So, a program that is generated using *offset* is likely to be sub-optimal in the presence of invalidations.

7.3.2 Auto-prefetch

The potential negative performance impact of invalidations is exacerbated if clients access the Broadcast Disk in a purely demand-driven manner. When important pages are invalidated, a demand-driven approach will fault them into a client's cache one-at-a-time. If *offset* is used, the performance penalty can be tremendous, as the most important pages have been placed on the slowest disks. This problem, however, can be easily addressed in the Broadcast Disk environment through the use of a new technique called *auto-prefetch*. As demonstrated in Chapter 6, the broadcast medium is in general, highly conducive to data prefetching; data pages continually flow past clients so the clients can choose to prefetch passing pages without imposing any additional load on shared resources. Auto-prefetch is a middle ground between invalidation and propagation (described next) – it tries to achieve the performance enhancement provided by propagation with the low overhead of invalidation.

Auto-prefetching is based on the intuition that with good cache management, the pages in a client's cache tend to be pages that are valuable to the client, whereas pages that are not cached tend to be less valuable. Auto-prefetching simply turns on prefetching at a client for any data pages that are invalidated from that client's cache. After a page is invalidated and marked for auto-prefetch, it is automatically brought into the cache the next time it appears on the broadcast. As will be shown in the performance experiments, auto-prefetching goes a long way towards mitigating the negative impact of invalidations.

7.3.3 Propagation

In contrast to invalidation, propagation delivers the new value of a changed data page, allowing a client to install that new value as a replacement for a cached value that would otherwise become stale. The advantage of propagation is that it can help clients stay closer to their steady state, especially when used in conjunction with auto-prefetch, as described above. The disadvantage of propagation is that it can be wasteful of bandwidth. When a page that is not cached

PrefetchOn	True
UpdateOn	True
BackChannelOn	False
NumClients	1
ThinkTime	2
UpdateThinkTime	2,5,10,20,25
CacheSize	100(Small), 500(Large)
θ, θ_u	0.95
Offset	0, *CacheSize*
UpdateOffset	0, 500

Table 7.1: Update Parameter Settings

at any clients (or cached at only very few clients) is propagated, the bandwidth used for that propagation effectively goes unused. Also, propagation upsets the natural broadcast schedule which may have to be suspended to propagate the new values.

In this experiments that follow, propagation is implemented through the use of a *propagation list*, which is simply a concatenation of new page values. As with invalidations, propagations can be saved up and broadcast periodically, or they can be broadcast immediately. In the system studied here, a propagation list contains the values of all pages that have been updated since the last propagation list was broadcast.

It is important to note however, that the dissemination of invalidations and propagations need not be done using the same policy. Correctness can be ensured through the timely broadcast of invalidations, in which case propagation is performed as a performance enhancement. Also, the maximum rate of propagations is significantly lower than that of invalidations, due to the differences in the bandwidth requirements — propagations cannot be transmitted any faster than the broadcast speed (in pages per unit time) of the medium.

As in traditional systems, there is an inherent tradeoff between propagation and invalidation. The tradeoffs are somewhat different in this environment, however, because of the dissemination-oriented nature of the medium and this is the focus of the studies in the next section.

7.4 Experiments and Results

7.4.1 Parameter Settings

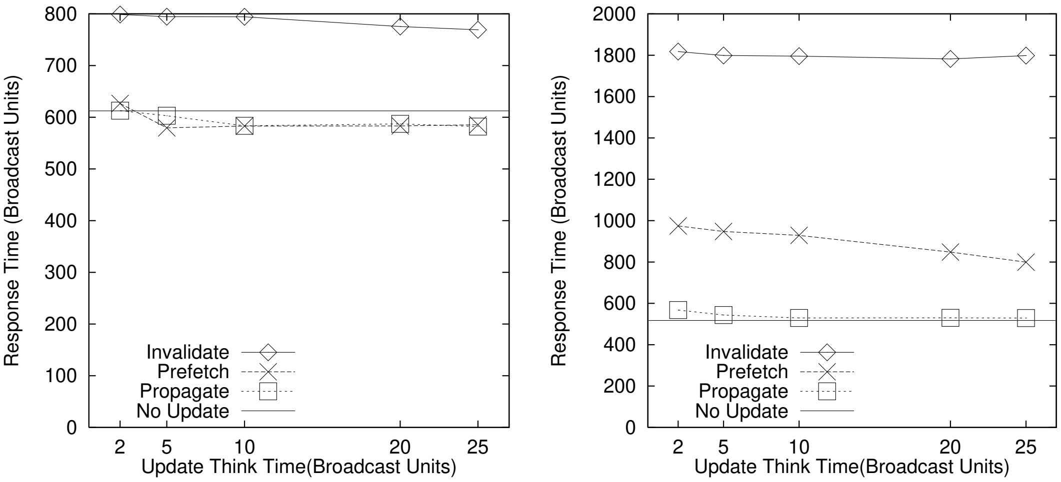
In this section, we use the simulation model to explore the influence of updates on the client performance. The details of the simulation model and the enhancements required to support updates are presented in Chapter 4. The primary performance metric is the average response time (i.e., the expected delay) at the client measured in *broadcast units*. Table 7.1 shows the parameter settings for these experiments. The basic database and broadcast parameters are as in Chapter 6 and the table lists only the parameters which are specific to the update study. The server database size was 3000 pages and the client access range was 1000 pages. The three-disk broadcast program from the last chapter was used for all the experiments. The size of the fastest disk was 300 pages, the medium disk was 1200 pages and the slowest disk was 1500 pages. The relative spin speeds of the three disks were 5, 3 and 1, respectively (i.e., Δ of 2). Two sizes for the client cache were used in the experiments — 100 and 500 pages. The results in these experiments were obtained once the client performance reached steady-state. The cache management policy is the $\mathcal{LIX}$ policy with the improved probability model (i.e., $\mathcal{LIX}(2)$ in Section 6.4.1). For the experiments in this chapter, a learning sample length of 10,000 access was chosen. The rest of the client and server parameters have the same values as the experiments in the last chapter.

We assume that an invalidation list does not cost any bandwidth and can be piggybacked on a regular page broadcast. This is a reasonable assumption because the size of such a list would be negligible compared to the size of a page. Every propagation of a new value, however, consumes a page worth of bandwidth, i.e., one broadcast unit.

If the server uses a propagation-based scheme to mitigate the effect of invalidations, the regular order of the broadcast is disturbed due to the infusion of the propagation list. The propagation list is broadcast by first suspending the regular broadcast, then transmitting the propagation list and then continuing the regular broadcast. Note that the propagation list only changes the absolute broadcast time of the pages (by the length of the list); the relative order among pages in the broadcast structure remains the same.

As stated in Section 7.3, updates and subsequent invalidations disturb the steady-state behavior of the client cache that we observed in read-only workloads. This section shows that we do not have to abandon the intuitions that were developed there. In particular, the $\mathcal{LIX}$ caching algorithm is still sound and the use of a broadcast offset can be retained without significant performance degradation. The update case, however, introduces new trade-offs that must be considered in order to stay close to the steady-state performance shown before. This section explores these tradeoffs.

We first study the client performance when updates are transmitted at the beginning of every major cycle. This is representative of the *Periodic* consistency model. The next set of experiments are for the case when the updates are disseminated immediately as they happen at the server and is representative of the *Latest Value* model for consistency. Finally, we describe refinements to the strategies proposed in Section 7.3 and do a sensitivity analysis in the presence of *Noise*. Recall that *Noise* is used as a mechanism to simulate disagreement between the server broadcast and the client needs due to the presence of multiple clients in the system.



(a)*Offset* = 0 (b) *Offset* = 100

Figure 7.1: Sensitivity to Updates Once a Cycle, *CacheSize* = 100, *UpdateOffset* = 0

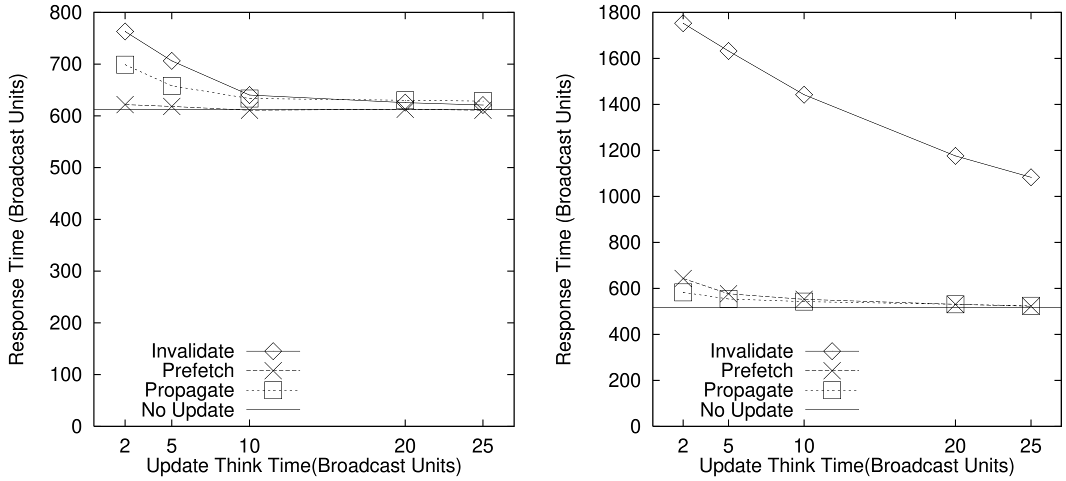
7.4.2 Major Cycle Update

First, we consider the case in which updates are communicated to the client at the start of a major cycle. As was mentioned earlier, this case is useful when the stability of values must be maintained. This case illustrates some very fundamental principles regarding the nature of the broadcast medium. The experiments in this section are for the case when there is no *Noise* in the system.

We begin by comparing three basic cache update techniques— *invalidate*, *prefetch*, which uses auto-prefetch (as described in Section 7.3) to re-acquire invalidated pages, and *propagate*, which combines propagation with auto-prefetch. For invalidate, the server sends a list of updated page numbers at the end of the cycle. For propagate, the server follows the invalidation list with the actual updated pages, and for prefetch, the server sends an invalidation list at the end of a cycle, and the client prefetches new values of the pages from the next major cycle.

Figure 7.1 shows a comparison of these three techniques for a cache of 100 pages when the client's hot pages (for read) are also the most frequently updated (*UpdateOffset* = 0). The graph on the left (Fig 7.1(a)) is for the case of no offset while the graph on the right (Figure 7.1(b)) represents the broadcast with an offset of *CacheSize*, i.e., the first 100 pages are shifted to the slowest disk. The x-axis, *UpdateThinkTime*, reflects the intensity of updates at the server. This is also measured in broadcast units, and is the number of pages broadcast per update posted at the server. Thus, the lower the *UpdateThinkTime*, more frequent the updates. For an update think time of 2, every alternate page is an update page (in the worst case). The solid line in all the graphs corresponds to the read-only case (an infinite update think time).

The first observation in Figure 7.1(a) is that the response time of invalidate is very poor.



(a) *Offset* = 0 (b) *Offset* = 100

Figure 7.2: Update does not Match Read Access, *CacheSize* = 100, *UpdateOffset* = 500

Here, a large number of pages are being removed from the cache (recall that *UpdateOffset* = 0) at the end of the cycle and they do not return until they are requested again. The invalidation severely pushes the client away from the steady-state case. The latency caused by having to fault in those missing pages one-at-a-time is so large that many of the benefits of caching are not realized and the performance suffers significantly.

Prefetch and propagate have approximately the same performance as the read-only case here, because the pages that are updated tend to come back before a new request arrives. In the case of propagate, the pages return in the propagation list which is broadcast immediately after the changes are communicated. Prefetch does as well as propagate for two reasons. First, since there is no offset, the client's hottest pages are on the fastest disk and thus, they are auto-prefetched without significant delay. Secondly, the cache is small and it is quickly restored to the state before the invalidations. Thus, prefetching is required in this case. Nothing is gained from additional propagation when the updated pages are on the fastest disk since the broadcast disk itself is acting as an effective propagation medium.

Before leaving Figure 7.1(a), we point out an interesting artifact of the $\mathcal{LIX}$ implementation. Notice that both invalidate and propagate do slightly better than the pure read-only case for *UpdateThinkTime* greater than 3 or 4. $\mathcal{LIX}$ approximates probabilities (as does LRU) and thus, sometimes makes incorrect decisions for choosing victims during page replacement. Since, in this case, the updates coincide with the access pattern, updated pages (the ones $\mathcal{LIX}$ should cache) are being pulled out of the chains and reinserted at the top because they are later auto-prefetched. Thus, the pages that are accessed infrequently drift to the bottom of the LRU chains faster, flushing them more quickly from the cache than in the no-update case.

Figure 7.1(b) shows the same case as Figure 7.1(a), but with an offset equal to the cache size of 100. Recall that offset is used to improve performance in the read-only case. Notice that here invalidate does even worse than before, but the performance of prefetch also deteriorates relative to propagate. At very high update rates, this difference is roughly an 80 to 90% degradation. By introducing an offset, the hot pages, which are also the most volatile pages, are being broadcast relatively infrequently since they are now on the slowest disk. The only way to improve this situation is to propagate the new values to counteract the slowdown introduced by the offset.

We next move to the case in which the read access does not match the update pattern. This is introduced by setting the *UpdateOffset* to 500 as in Figure 7.2. Consider Figure 7.2(a) in which *Offset* is zero. For lower update rates (the right-hand end of the graph), invalidate is similar to propagate for three reasons:

1. Not many cached pages are invalidated.
2. Most of the cached pages are from the fast disk and thus have a low latency.
3. The propagation list is small.

For the first two reasons, prefetch alone move the client back to the read-only scenario, much as it did in Figure 7.1(a). The number of pages that must be re-acquired is small and the overhead of a large propagation list is not cost effective. As we move to the left, thereby increasing the update rate, the first and third reasons are no longer true. More cached pages are invalidated and the propagation list is longer.

Notice that in Figure 7.2(a), prefetch is better than propagation when the update rate is high. A fundamental principle at work here is that propagation only helps when the server propagates pages that the client needs. If the propagation includes pages that are not accessed, performance degrades because of the wasted bandwidth. For items that are not requested often or that are broadcast frequently, it is good enough to allow them to drift back into the cache through prefetch.

As we move to a larger cache size, the cached pages will come from other than the fastest disk (point #2 above). Recall that the size of the first disk is 300 pages. When this happens, the need for propagation becomes greater. The results for the large cache case are qualitatively similar to Figure 7.2 with prefetch doing somewhat worse.

Figure 7.2(b) introduces an offset of cache size (i.e., 100). Here, the performance of invalidation is again extremely poor since the hottest pages are on the slowest disk. Thus, point #2 in the above list is not true for this case. Prefetch and propagate perform well because they are both returning updated pages to the cache fast enough to keep up with demand. This differs from the the analogous *UpdateOffset*=0 case (Figure 7.1(b)) in that the number of hot pages that are being updated is small. Thus, prefetch is good enough (except at very high update rates).

In this set of studies, for the update once per major cycle case, we have shown that there exist algorithms that can compensate for potential performance loss due to updates in the system. The client, however, has to prefetch invalidated pages in order to do well. In general choosing propagation in this context produces good results, although for *UpdateOffset* = 500, the client can do almost as well with prefetch alone. Given the extremely poor performance of the invalidate strategy, we will no longer discuss its performance in the following sections.

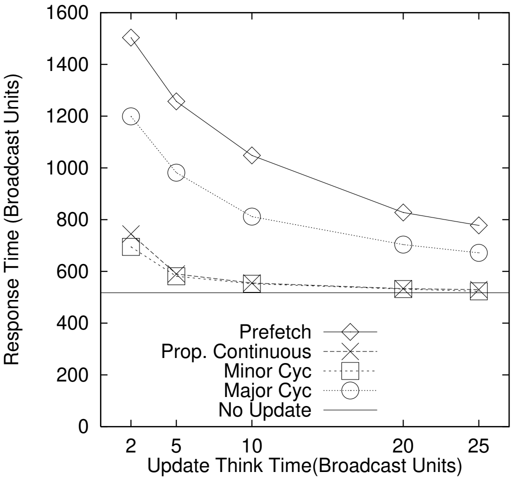
As stated earlier, the best broadcast for steady-state situations is one in which the hottest *CacheSize* pages are offset onto the slowest disk. The benefit of an offset is visible in the lower response time (by about 16%) for the no update case in Figure 7.1(b) (*Offset*=100) over Figure 7.1(a) (*Offset*=0). We want the Broadcast Disk system to perform well for both high and low update periods and for clients that access volatile and non-volatile data. In order for the broadcast to provide clients with the best performance possible, we strive to retain the offset in the broadcast program and implement strategies which mitigate the effects of updates. Thus, we will only consider the case of a *CacheSize* offset in the experiments that follow.

7.4.3 Continuous Update

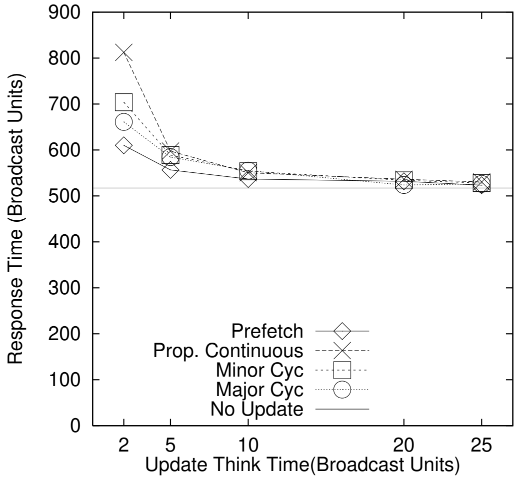
In this section, we study an alternative consistency model in which updates are communicated to the clients as soon as they are posted at the server. Since, as explained above, only broadcasts with an offset are being considered, the server must propagate; however, the issue is how often, since the server is no longer constrained to propagate only once per major cycle. The options we consider are a) propagate continuously (immediately following invalidations) b) propagate at minor cycle boundaries (although this is an arbitrary intermediate point) and c) propagate at major cycle boundaries.

The four graphs in Figure 7.3 illustrate the client performance for various propagation rates for small and large caches and for an *UpdateOffset* of 0 and 500. In all of these cases, continuous propagation (legend: cross) never beats minor cycle propagation (legend: box). The lesson here is that it is possible to propagate too often and consequently, waste bandwidth. In other words, it is best to propagate at a rate that just keeps up with demand— less often causes expensive faults, and too often wastes bandwidth. We will study this tradeoff in more depth in what follows.

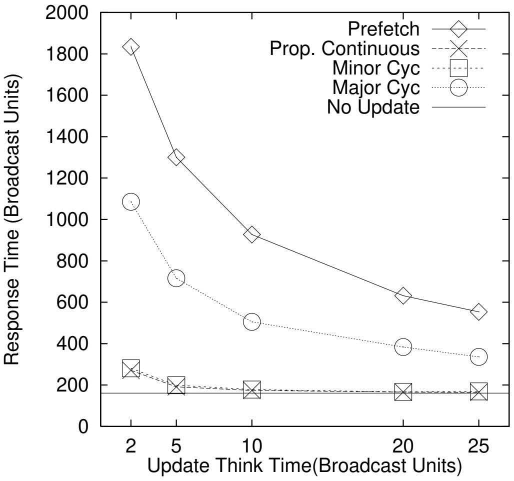
The two graphs in the left-hand column of Figure 7.3 show the results for a small (Figure 7.3(a)) and a large (Figure 7.3(c)) cache with an *UpdateOffset* of zero. In both of these cases, because items of interest to the client are being updated, the server must propagate more than once per major cycle. Notice the large performance gap between major cycle propagate and minor cycle propagate. Surprisingly, there is little difference between minor cycle propagate and continuous propagate. In fact, with a small cache (Figure 7.3(a)) and a high update rate (e.g., *UpdateThinkTime* = 2), minor cycle propagate does slightly better than continuous propagate. This is because minor cycle propagation gets updates to the client fast enough and saves bandwidth by propagating a page that is updated multiple times within a minor cycle only



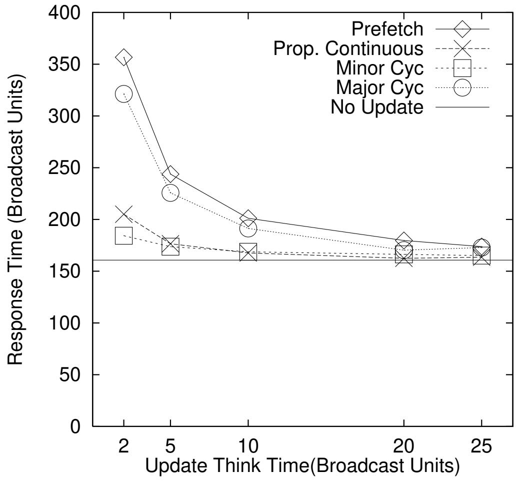
(a) *CacheSize* = 100, *UpdateOffset* = 0



(b) *CacheSize* = 100, *UpdateOffset* = 500



(c)*CacheSize* = 500, *UpdateOffset* = 0



(d) *CacheSize* = 500, *UpdateOffset* = 500

Figure 7.3: Response Time vs. Continuous Update for Various Cache Sizes and *UpdateOffset*

once. The additional bandwidth consumed in continuous propagate by repeatedly propagating the same page simply slows down the regular broadcast.

An *UpdateOffset* of 500 is introduced for both the graphs in the right-hand column of Figure 7.3. For a large cache (cache size = 500), minor cycle propagation is still preferred over continuous propagation (Figure 7.3(d)). Even though a smaller percentage of cached pages are changing, a larger cache means that there are still a significant number of pages that are being invalidated. The penalty of not getting these pages back before they are accessed makes major cycle propagation unresponsive.

The small cache case (cache size = 100) with an *UpdateOffset* of 500 is the only case in which minor cycle propagation does not win(Figure 7.3(b)). Here, in the case of very high update rates, a pure prefetch scheme is preferable but only by about 10%. When the update does not match the read-access pattern and the cache is small, propagating the changes degrades performance since it uses bandwidth for uninteresting pages. The large number of propagated pages, even at low rates such as once per major cycle, is enough to slow down the normal broadcast.

In summary, minor cycle propagation does best in all cases except for a small cache and a high update rate and when there is a large discrepancy in the update and the access pattern. In this case, prefetch alone performs better but only by a small amount. Thus, for most cases minor cycle propagation is the technique of choice.

7.4.4 Refining the Results

In the previous section, we showed that under most circumstances, a propagation algorithm that strikes a balance between infrequent (once a major cycle) and continuous propagation is the most desirable. Propagating at minor cycle boundaries is one example of such a compromise. In this section, we investigate the possibility of propagating a "subset" of the changes as a way to ameliorate the waste of bandwidth introduced by frequent propagation. If only the *good bets* are propagated frequently, it might be possible to improve the results.

We study three variations on this theme. Each of these algorithms uses some static or dynamic information to filter the number of pages propagated. The first, called *Server_Offset*, propagates only the server's offset pages. These are the pages which are the hottest (for read) from the server's viewpoint. Recall that the server uses a probability distribution to generate the broadcast which is a cumulative function over the access patterns of all the clients in the system. When there is no *noise* in the system, the server's distribution is exactly the same as the client's distribution (and so are their offset pages). However, as *noise* increases the client's access distribution drifts away from the server broadcast and the overlap between the offset pages and the client's hot pages decreases, thereby, reducing the utility of the propagation list.

The next algorithm, called *Slow_Disk*, propagates updates only if they reside on the slowest

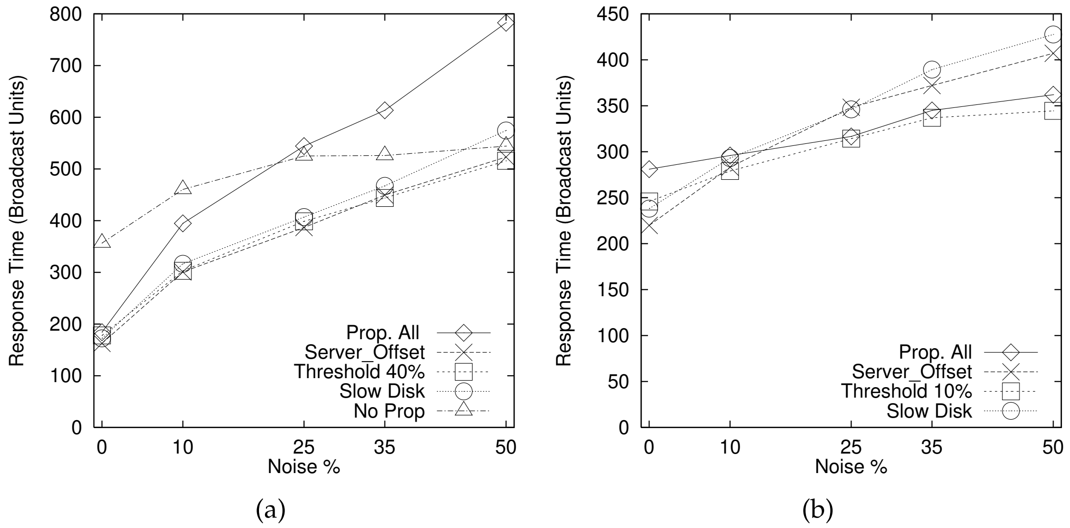


Figure 7.4: *Noise* Sensitivity, *CacheSize* = 500, (a)*UpdateOffset* = 500, (b) *UpdateOffset* = 0

disk. This is a simple strategy which requires no probability information. Its goal is to propagate the pages which, when invalidated, hurt the client most. Note that for the parameter settings under consideration, it propagates at least as many pages as the *Server_Offset* algorithm. The last algorithm, called *Threshold*, propagates a page only if its next arrival time on the broadcast is greater than a fixed threshold. The threshold is chosen as some percentage of the major cycle size. It uses a dynamic metric and thus, is potentially expensive to implement.

We performed a sensitivity analysis on the value of the propagation threshold. It was found that for an *UpdateOffset* of zero, a propagation threshold of 10% of a major cycle provides the best response for most values of *noise*. For an *UpdateOffset* of 500, the best threshold value was around 40%. When the write and the read access coincide (*UpdateOffset*=0), the threshold is low because it is likely that an updated page will be accessed before it is broadcast again. We use these best cases in the graphs that follow. The results in this section consider the case when pages are propagated once a minor cycle (with invalidations occurring continuously).

Figure 7.4 shows the performance of all the algorithms for varying *noise* (on x-axis) for a cache size of 500 for the most update intensive case studied earlier (*UpdateThinkTime* = 2) and for two values of *UpdateOffset*. *Noise* represents variance in the access patterns of the clients. The key to good performance in steady-state, is to give priority for cache real-estate to those pages which are *hot* for the client locally but not globally popular and thus, not broadcast often. Other pages are accessed off the broadcast. In the read-only case, as long as the client has enough cache space for such locally important pages, it can maintain its steady-state performance even as the *noise* increases (up to a point). When there are updates, the cache never quite reaches steady-state since pages are being invalidated often. *Noise* only makes this worse, since with increasing variance among the client access patterns (as represented by *noise*), fewer of the clients' locally popular pages are also globally popular. The two graphs in Figure 7.4 show this decline in performance as *noise* increases.

We see that no algorithm performs uniformly well. The *Server_Offset* algorithm is by far the winner in most cases. It does quite well in Figure 7.4(a) in which *UpdateOffset*=500. *Server_Offset* is a conservative policy— it only propagates pages that are known to be hot from the server's point of view (the server's offset pages). Since the server distribution is generated as an average over all the client access patterns, it implies that the majority of clients want these pages. The best *Threshold* case does as well as *Server_Offset* with the *Slow_Disk* performing slightly poorer. The folly of aggressively propagating every updated page is apparent from the crossover of no propagate (i.e., prefetch alone) at around 25% *noise*.

Server_Offset is also a good bet for *UpdateOffset*=0 (Figure 7.4(b)) when the *noise* is low. As the *noise* increases beyond 10%, *Server_Offset* begins to do worse. With increasing noise, some of the pages that have high access for the client and high update no longer correspond to the server's offset pages. Thus, *Server_Offset* does not propagate them, and they do not come by fast enough. In fact, for *UpdateOffset*=0 as *noise* increases, propagating all pages performs very well. Since most of the updated pages are accessed by the client, the extra bandwidth is well spent. The best *Threshold* case (threshold of 10%) again performs the best when the *noise* is high. However, the inability to find a threshold value which does uniformly well over the complete simulation space makes *Threshold* a less than viable option in its current form. An algorithm which dynamically adjusts the threshold value can be expected to do well, and developing such strategies is an avenue for future work. We do not show the performance of the no propagate case here because of its extremely poor performance (about 5 times worse than the others).

In general, *Server_Offset* is a good refinement to propagating everything. It does well except in cases in which the update and the read access coincide and in which the *noise* level is high. Note that even here the loss is minimal when compared to prefetch with no propagation. We would argue that high *noise* is not that interesting for us. In order to maintain a Zipf distribution at the server, there cannot be a wide variance in the client's behaviors. If the variance were high, the combined distribution at the server would tend more toward uniformity.

7.5 Related Work

Influence of updates on performance in dissemination-based environments has been studied in [ABGM90, BGH$^+$92, BI94, JBEA95, WYC96]. In [ABGM90], the notion of *quasi-copies* was introduced where a replica was allowed to deviate from the original copy in a controlled fashion. In this environment, consistency could be defined on a per-client basis. Based on the constraints supplied, the client would be guaranteed to have the access to a value which was up-to date within the last t minutes of the most recent value or within $n\%$ of the current value (for scalar variables) and so on. The Datacycle Project [BGH+92] allowed clients to execute queries or transactions locally and use the upstream network to send updates back to the server. The system provided consistency by ensuring that data items read within a single broadcast cycle were mutually consistent. For queries spanning over multiple cycles, the client used an optimistic protocol and the server broadcast a log of changes which was used to detect conflicts. This periodic consistency guarantee is similar to our once-a-major-cycle update technique.

[BI94] addressed the problem of updates and consequently, stale data in a client's cache in a mobile setting where clients could be disconnected for long periods of time. Among other ideas, they proposed periodic broadcast of invalidations using signature based schemes. [JBEA95] and [WYC96] are recent papers which improve upon the algorithms proposed in [BI94]. These papers differ significantly from ours, however, because they assume a *pull-based* system where the client fetches data by making explicit requests to the server via the back channel; only the invalidations are disseminated. The techniques developed in this area, however, may be useful for supporting intermittent connectivity in the Broadcast Disk environment as well.

Finally, a discussion and comparison of the caching algorithms (and techniques to cope with updates) in traditional client-server systems is available in [Fra96]. For similar surveys in other areas the reader is referred to [LS90] (distributed file systems), [NL91] (distributed shared memory) and [AB86] (multi-processor architectures).

7.6 Conclusions

In this chapter, we examined the influence of updates on the client performance in the Broadcast Disks framework. As in the last two chapters on cache management, the design considerations divide into client-side and server-side issues. The server must decide how to modify its broadcast program in order to most effectively communicate updates to the client. The client must perform appropriate actions (e.g., prefetch) to bring its cache contents back to steady-state. For the server, we experimented with invalidation and propagation lists. At the client, our experiments examined cache invalidation and prefetch.

Two different consistency models were studied — immediate and major-cycle boundary representative of the *latest value* and the *periodic* models respectively. We showed that for low to moderate update rates, it is possible to approach the performance of the read-only case.

The experiments demonstrated two fundamental differences with traditional pull-based client-server systems. The first difference was that invalidations and propagations can happen at different rates without hurting performance. For example, the server could use an immediate invalidation strategy along with a minor cycle propagation strategy. By invalidating immediately, the server preserves the correctness semantics at the client and by batching propagations it saves bandwidth which offsets the delay clients may experience in accessing invalidated pages. Traditional systems rarely de-couple invalidations and propagations and the studies in this chapter point to the efficacy of such an approach.

The second difference from traditional systems is the superiority of propagation-based strategies in general over invalidation-based strategies. In traditional client-server systems, the converse is usually true with invalidation-based strategies dominating in performance [Fra96]. This reversal is due to the differences in tradeoffs for the two schemes. In this environment, due to the broadcast nature of the link, a single propagation message suffices in informing all the interested clients of the updated value. In a client-server system, the server has to individually propagate the new values to all clients with a cached copy of the updated page. This effort not only wastes bandwidth but also may be futile if most of the clients do not care about the new value.

We also showed that in some cases prefetching is often good enough. We demonstrated that propagating a page is useful if the client makes use of the page before it would naturally get it back by prefetch or direct access. Otherwise, the propagation wastes bandwidth. This is one of the fundamental tradeoffs in the design of dissemination-based systems with update.

A key observation from these experiments in this chapter is that, in general, the Broadcast Disks model is robust in the presence of updates. We have shown that with some very simple enhancements it is possible to produce results that closely track the read-only case. Thus, all the intuitions and results that were observed for the read-only case (Chapters 5 and 6) are still valid in the update setting.

Chapter 8

Balancing Push and Pull

8.1 Introduction

The last few chapters have investigated the Broadcast Disks paradigm for the extreme case of asymmetric communication, i.e., clients only listen in to the broadcast without an uplink channel to connect back to the server. In this chapter, we extend this previous work by integrating an uplink channel (or, backchannel) into the Broadcast Disks model. This study is architecturally different from each of the previous chapters due to the presence of a backchannel. Thus, before delving into the issues at hand, we will summarize the earlier push-only Broadcast Disks approach to put this study in context.

The Broadcast Disks approach is a push-based technique for data dissemination. Here, the server uses its best knowledge of client data needs (e.g., as provided by client profiles) to construct a broadcast schedule containing the items to be disseminated and repetitively transmits this "program" to the client population. Clients monitor the broadcast and retrieve the items they require (i.e., those that may be needed locally but are not currently cached in local storage) as they come by. In such a system, data items are sent from the server to the clients without requiring a specific request from the clients.

In contrast to a periodic broadcast approach, traditional client-server database systems and object repositories transfer data using a *pull-based*, request-response style of operation (e.g., RPC). Using request-response, clients explicitly request data items by sending messages to a server. When a data request is received at a server, the server locates the information of interest and returns it to the client. Pull-based access has the advantage of allowing clients to play a more active role in obtaining the data they need, rather than relying solely on the schedule of a push-based server. However, there are obvious disadvantages to pull-based access as described in Section 2.2.1. We briefly list them here again:

- *Backchannel Requirement.* A basic necessity of any pull-based system is an uplink channel for the clients to make requests to the server. In some environments, such as wireless networks and CATV, the provision of a backchannel can impose substantial additional expense or may be absent altogether.

- *Backchannel congestion.* In many emerging environments, the backchannel is limited in bandwidth and can easily become a bottleneck.

- *Server saturation.* In a pull-based system, the server must be continually interrupted to react to the client backchannel requests and can easily become a scalability bottleneck with large client populations. Saturation is informally defined as the state when the rate of requests arriving at the server is greater than the rate at which it can service them. Thus, as the number of clients increase, eventually the server will become saturated.

In this study, the clients have access to a backchannel which they use to *pull* pages that are not available in the client cache and that will not appear quickly enough in the broadcast. The goal of doing so is to improve client responsiveness by making the server broadcast the page "out-of-turn" before its scheduled transmission. Besides pulling misses, there are a number of other potential uses of a backchannel for clients in a push-based system. Examples include:

- *Send feedback and new profiles.* The backchannel can also be used to provide feedback to the server if it has setup a poorly designed broadcast or to send in a new profile if the client access pattern changes.

- *Allow local updates.* Once clients have a backchannel they can propagate back to the server any updates they make to the their locally cached copies. This is extremely important if they system intents to provide transactional support.

- *Re-broadcast corrupted data.* If the transmission channel is error-prone, client can use the backchannel to request the server to re-transmit a page corrupted due to noise on the channel.

These alternative uses are interesting avenues for further study; however, they are beyond the scope of this thesis.

This chapter studies the impact of augmenting the Broadcast Disks model with a backchannel from the client to the server on the client performance. We focus on the tradeoffs between the push and pull approaches and investigate ways to efficiently combine them in order to mitigate their individual drawbacks. In this chapter, we attempt to address the following issues:

- *Scalability and performance.* We study the impact of client requests on steady-state and warm-up performance and investigate the scalability of this performance as the client population grows.

- *Balancing push and pull.* The introduction of a backchannel requires the server to divide the downstream bandwidth between the regular push schedule and the responses to the pull requests. In this study, we investigate the ramifications of varying the ratio of the bandwidth allocated to the push and pull delivery.

- *Postponing server saturation.* As pointed out in Section 2.2.1, it is always possible to swamp a server by allowing more clients to join the system than what it is designed for. In many dissemination-based environments (e.g., internet), it may be impossible to control the size of the client population. Thus, in the experiments that follow, we will investigate how to maximize the number of supported clients before the point of saturation is reached.

- *Influence of Noise.* In this chapter, we will also study if the presence of a backchannel can help reduce the sensitivity of the client performance to the variance in client access patterns (as defined by *Noise*).

- *Broadcasting database subset.* The final set of experiments study the impact of placing only a subset of the database on the Broadcast Disk and letting the clients use the backchannel to pull the rest of the database. This was not possible earlier, because in the absence of a backchannel, the server was forced to broadcast the entire data of interest to the client population.

The outline of the rest of this chapter is as follows. The next section discusses some of the new issues that arise when trying to integrate a backchannel. Also, the algorithms whose performance are compared will be introduced. Section 8.3 presents the various experimental studies. First, the basic tradeoffs of push and pull will be investigated. Then, Section 8.3.3 discusses techniques to improve scalability by reducing the use of the backchannel. The last set of experiments investigate the pros and cons of broadcasting only a subset of the database. Finally, Section 8.3.5 summarizes the results.

8.2 Integrating a Backchannel

In this chapter, we introduce a backchannel into the Broadcast Disk environment. The details of the simulation model incorporating a backchannel, on which the experiments of this chapter are based, was presented in Section 4.2.3. In this paragraph, we briefly summarize it again. We model the backchannel as a point-to-point connection with the server. Thus, the rate at which requests arrive at the server can grow proportionally with the number of clients. Note that since request messages are typically small, this assumption is justifiable even in situations where clients share a physical connection with the server. The server, on the other hand, has a maximum rate at which it can transmit pages in response to client requests (as described below), and moreover, it has a finite queue in which to hold outstanding requests (as measured by *ServerQSize*). If a request arrives when the queue is full, that request is thrown away.[1] The server queue is serviced in a first-come-first-serve (FCFS) order.

In order to support both push-based and pull-based delivery, the server interleaves pages from the broadcast schedule (i.e., push) with pages that are sent in response to a specific client request (i.e., pull). The percentage of slots dedicated to pulled pages is a parameter that can be varied. We call this parameter *pull bandwidth* (*PullBW*). When *PullBW* is set to 100%, all of the broadcast slots are dedicated to pulled pages. This is a *pure-pull* system. Conversely, if *PullBW* is set to 0%, the system is said to be a *pure-push* system; all slots are given to the periodic broadcast schedule, and thus, there is no reason for clients to make pull requests.

Setting *PullBW* to a value between these two extremes allows the system to support both push and pull. For example, if *PullBW*=50%, then at most, one page of pull response is sent for each page of the broadcast schedule. We define *PullBW* to be an upper bound on the amount of bandwidth that will be given to pulled pages. In the case in which there are no pending requests we simply continue with the Broadcast Disk schedule, effectively giving the unused pull slots back to the push program. This approach saves bandwidth at the cost of making the layout of the broadcast less predictable.[2]

It is important to note, that unlike the pure-push case, where the activity of a single client does not directly affect the performance of other clients, in this environment, overuse of the backchannel by one or more clients can degrade performance for all clients. There are two stages of performance degradation in this model. First, because the rate at which the server can respond to pull requests is bounded, the queue of requests can grow, resulting in additional latency. Second, because the server queue is finite, extreme overuse of the backchannel can result in requests being dropped by the server. The intensity of client requests that can be tolerated before these stages are reached can be adjusted somewhat by changing the *PullBW* parameter.

8.2.1 Algorithms

In the remainder of this chapter, we compare the performance of pure-push, pure-pull, and an integrated push/pull algorithm, all of which use broadcast. All three approaches involve client-side as well as server-side mechanisms. In all of these techniques, when a page is needed, the client's local cache is searched first. Only if there is a cache miss does the client attempt to obtain the page from the server. The approaches vary in how they handle cache misses.

[1] The server will also ignore a new request for a page that is already in the request queue since the processing of the earlier message will also satisfy this new request.

[2] Predictability may be important for certain environments. For example, in mobile networks, predictability of the broadcast can be used to reduce power consumption [IVB94a].

1. *Pure-Push*. This is the standard delivery model of the Broadcast Disks mechanism. Here, all broadcast bandwidth is dedicated to the periodic broadcast (*PullBW*=0%) and no backchannel is used. On a page miss, clients simply wait for the desired page to appear on the broadcast.

2. *Pure-Pull*. Here, all broadcast bandwidth is dedicated to pulled pages (*PullBW*=100%) so there is no periodic broadcast. On a page miss, clients immediately send a pull request for the page to the server. This is the opposite of *Pure-Push*, that is, no bandwidth is given to the periodic broadcast. It is still a broadcast method, though, since any page that is pulled by one client can be accessed on the frontchannel by any other client. This approach can be referred to as *request/response with snooping*.

3. *Interleaved Push and Pull* ($\mathcal{IPP}$). This algorithm mixes both push and pull by allowing clients to send pull requests for misses on the backchannel while the server supports a Broadcast Disk plus interleaved responses to the pulls on the frontchannel. As described previously, the allocation of bandwidth to pushed and pulled pages is determined by the *PullBW* parameter.

A refinement to $\mathcal{IPP}$ uses a fixed threshold to limit the use of the backchannel by any one client. The client sends a pull request for page p only if the number of slots before p is scheduled to appear in the periodic broadcast is greater than the threshold parameter called *ThresPerc*. Threshold is expressed as a percentage of the major cycle length (i.e., the push period) as described in Section 3.2. When *ThresPerc*=0%, the client sends requests for all missed pages to the server. When *ThresPerc*=100% and the whole database appears in the push schedule, the client sends no requests since all pages will appear within a major cycle.[3] Increasing the threshold has the effect of forcing clients to conserve the use of the backchannel and thus, minimize the load on the server. A client will only pull a page that would otherwise have a very high push latency. In effect, threshold makes the clients pull only their *most expensive* misses.

The $\mathcal{PIX}$ cost-based algorithm, is used as the cache management strategy for the *Pure-Push* and $\mathcal{IPP}$ algorithms. Since there is no concept of a cyclic broadcast when using *Pure-Pull*, the $\mathcal{P}$ algorithm is used for page replacement in that case. Both $\mathcal{P}$ and $\mathcal{PIX}$ have been introduced earlier in Chapter 5.

In Section 8.3, we examine the performance of these different approaches, as well as the impact of parameters such as *PullBW* and *ThresPerc* (in the case of $\mathcal{IPP}$). We also investigate the pull-based policies for cases in which only a subset of the database is broadcast. In particular, we examine the performance of the system when the slowest and the intermediate disk in the broadcast program are incrementally reduced in size.

[3]The case where a server disseminates only a subset of the pages is studied in Section 8.3.4.

PrefetchOn	True
UpdateOn	False
BackChannelOn	True
NumClients	2
CacheSize	100
ThinkTime	20
ThinkTimeRatio	10, 25, 50, 100, 250
SteadyStatePerc	0%, 95%
ServerDBSize	1000
NumDisks	3
$DiskSize_{1,2,3}$	100, 400, 500
Δ	1
ServerQSize	100
PullBW	10%, 20%, 30%, 40%, 50%
ThresPerc	0%, 10%, 25%, 35%

Table 8.1: Backchannel Parameter Settings

8.2.2 Assumptions

The general assumptions about the environment have been listed in Section 3.4. In addition to those, the experiments in this chapter also make the following two more:

1. *Independence of frontchannel and backchannel.* In our model, both the backchannel and the front channel are physically separate and the traffic on either of the channels do not interfere with each other. While this is assumption may not always be valid for some network technologies like Ethernet which share the channel, it is true of many others as in CATV and satellite networks.

2. *Data is read only.* In the last chapter, we studied techniques for managing volatile data in a Broadcast Disk environment. We showed that, for moderate update rates, it is possible to approach the performance of the read-only case. Thus, in order to make progress on the problem at hand, we have ignored that issue in this study.

8.3 Experiments and Results

In this section, we use the simulation model to explore the trade-offs between using push and pull to access data in a broadcast environment. The primary performance metric, as in the earlier experiments, is the average response time (i.e., the expected delay) at the client measured in *broadcast units*.

8.3.1 Parameter Settings

Table 8.1 shows the parameter settings for these experiments. The simulation model has been described in Chapter 4 and the specific extensions for supporting a backchannel is in Section 4.2.3. The server database size (*ServerDBSize*) is 1000 pages and a three-disk broadcast is used for all the experiments. The size of the fastest disk is 100 pages, the medium disk is 400 pages and the slowest disk is 500 pages. The relative spin speeds of the three disks are 3, 2 and 1, respectively (i.e., Δ=1). The client cache size is 100 pages (10% of the database). The backchannel queue can hold up to 100 requests for distinct pages, and the percentage of slots on the broadcast allocated to pull pages for $\mathcal{IPP}$ (*PullBW*) is varied from 10% to 50%. Recall that for these set of experiments, the simulator needs to run two clients $\mathcal{MC}$ and $\mathcal{VC}$ to simulate the presence of multiple clients. This is achieved by running both the clients at various speeds as determined by their think time values. The $\mathcal{MC}$_*ThinkTime* is set to 20 broadcast units, and the *ThinkTimeRatio* is varied so that the $\mathcal{VC}$ generates requests from 10 to 250 times more frequently than the measured client. This can be thought of as placing an additional load on the server that is equivalent to a client population of *ThinkTimeRatio* clients operating at the same rate as the $\mathcal{MC}$. The rest of the client and server parameters take on the same values as in the experiments of the last chapter. Except where explicitly noted, the response time results were obtained once the client reached steady state.

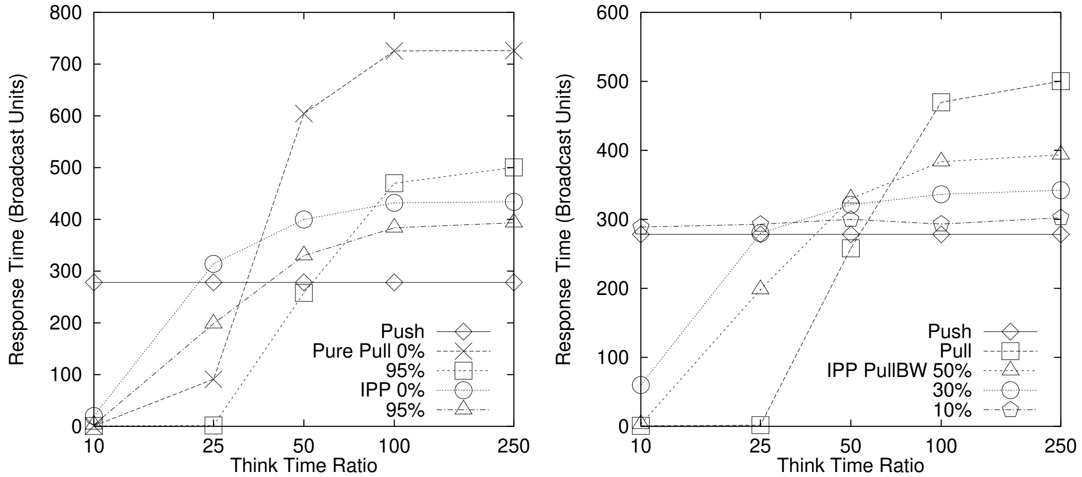
8.3.2 Experiment 1: Basic Tradeoffs

In the first experiment, we examine the basic tradeoffs between a pure push-based approach, a pure pull-based approach, and $\mathcal{IPP}$, which combines the two in a straightforward manner as described in Section 8.2.1.

Steady-State Performance

Figure 8.1(a) shows the expected average response time of requests made by the $\mathcal{MC}$ once steady-state has been reached versus the *ThinkTimeRatio*. In this case, for $\mathcal{IPP}$, *PullBW* is set to 50%; that is, up to half the broadcast bandwidth is dedicated to broadcasting pulled pages (i.e., those requested on the backchannel). Of course, in the *Pure-Push* and *Pure-Pull* case, the entire broadcast bandwidth is dedicated to pages on the periodic broadcast and to the pulled pages respectively.

Five curves are shown. The performance of *Pure-Push* is a flat line at 278 broadcast units. *Pure-Push* does not allow clients to use the backchannel, so its performance is independent of the size of the client population (or *ThinkTimeRatio*). For each of the two pull-based approaches (i.e., *Pure-Pull*, and $\mathcal{IPP}$), two separate curves are shown. These curves, labeled as 0% and 95%, show the cases in which the $\mathcal{VC}$'s access pattern is generated as if 0% and 95% of the clients it represents are in the steady-state, respectively (i.e., *SteadyStatePerc* = 0% or



(a) $\mathcal{IPP}$ *PullBW*=50%, *SteadyStatePerc* Varied (b) $\mathcal{IPP}$ *PullBW* Varied, *SteadyStatePerc*=95%

Figure 8.1: Steady State Client Performance

95%). Clients in steady-state are assumed to have completely warmed-up caches containing the highest valued pages.[4]

In general, for the pull-based schemes, better steady-state performance is achieved when most of the other clients are also in the steady-state (e.g., the 95% curves). This behavior is to be expected — a client is likely to have better performance if its needs from the broadcast are similar to the needs of the majority of the client population. If so, it can "snoop" the responses of requests from other clients for pages of common interest. Clients that are in the warm-up state (or that do not have caches) are likely to request the highest valued pages and thus, a considerable amount of the bandwidth dedicated to pulled pages is likely to be taken up by such pages. In contrast, once clients are in steady-state, they do not access the $CacheSize - 1$ highest valued pages from the broadcast (they are obtained from the client cache), which frees up bandwidth for the next-most important pages. In this experiment, since the $\mathcal{MC}$ performance is being measured only after it has reached steady-state, it gains more benefit from the pulled pages when most of the other clients (i.e., the $\mathcal{VC}$) are in steady-state as well.[5]

At the left side of the graph, since the system is very lightly loaded, the requests sent to the server via the backchannel are serviced almost immediately. At the extreme left (*ThinkTimeRatio* = 10) all of the requests made by the clients are serviced quickly. In this case, the pull-based

[4]For *Pure-Pull* the measure of value is $\mathcal{P}$ (i.e., probability of access p), while for $\mathcal{IPP}$ it is $\mathcal{PIX}$ (i.e., p divided by the frequency of occurrence in the push broadcast schedule).

[5]This result holds only to the extent that clients have similar access patterns. The impact of differing access patterns is addressed in Section 8.3.2.

approaches perform similarly and several *orders of magnitude* better than *Pure-Push*.[6]. In a very lightly loaded system, pull-based approaches are able to provide much more responsive, on-demand access to the database, and since our model does not impose any costs for uplink usage, there is no incentive for a push-based approach in this scenario.

As the load on the server is increased (i.e., increasing *ThinkTimeRatio*), however, the performance of the pull-based approaches begins to degrade. Beyond a *ThinkTimeRatio* of 50, all of the pull-based approaches perform worse than *Pure-Push*, and when the system becomes even more heavily loaded, (e.g., at a *ThinkTimeRatio* of 100 and higher) *Pure-Pull* performs worse than both *Pure-Push* and $\mathcal{IPP}$. When all other clients are in the warm-up phase, $\mathcal{MC}$'s performance deteriorates sharply since the server is more liable to drop requests. Because of the disagreement between the needs of the $\mathcal{MC}$ and those of the rest of the clients, the $\mathcal{MC}$ pays a high penalty when one of its requests gets dropped, because the other clients have a lower probability of accessing the same page. Even when *SteadyStatePerc* = 95%, the performance of *Pure-Pull* deteriorates quite rapidly as the server queue fills. The $\mathcal{IPP}$ approach, which in this case, dedicates half of the broadcast bandwidth to pushing pages, has worse performance than *Pure-Pull* at low to moderate contention rates, but levels out to a better response time than *Pure-Pull* when the contention at the server is high. At lower loads, the pushed pages occupy broadcast slots that could better be used to service pull requests, while at higher loads, they limit how long a client will have to wait for a page, even if its request is dropped by the server.

In effect, while push has a poorer response time than pull at lower loads, it provides an upper bound on the latency for any page, i.e., a "safety-net", when the server congestion causes pull requests to be dropped. This is the fundamental tradeoff between push and pull in the Broadcast Disks environment.

This experiment, thus, demonstrates that in steady state if the system is always underutilized, *Pure-Pull* is the algorithm of choice and if the system is saturated, the server should disseminate via *Pure-Push*. However, the use of either of the two algorithms in a system with widely varying loads can lead to very erratic performance. The goal of this study is to develop strategies which provide consistently good response times over a wide range of system loads. The $\mathcal{IPP}$ algorithm is one such technique and as Figure 8.1(a) shows, while there are workloads for which *Pure-Pull* and *Pure-Push* do better, the performance of $\mathcal{IPP}$ is more uniform over the entire space. In the rest of this chapter, we will study three parameters that influence $\mathcal{IPP}$'s performance — pull bandwidth, threshold (as determined by *ThresPerc*) and the size of the push schedule. These sensitivity results highlight the system tradeoffs and provide input to fine-tune $\mathcal{IPP}$ for consistent performance across a broad spectrum of server loads.

[6] $\mathcal{IPP}$ with *SteadyStatePerc* = 0% performs somewhat worse than the others at this point. Since half of the broadcast bandwidth is dedicated to push, much of the remaining bandwidth is consumed by non-steady-state clients pulling pages that the $\mathcal{MC}$ already has cached.

Impact of Pull Bandwidth

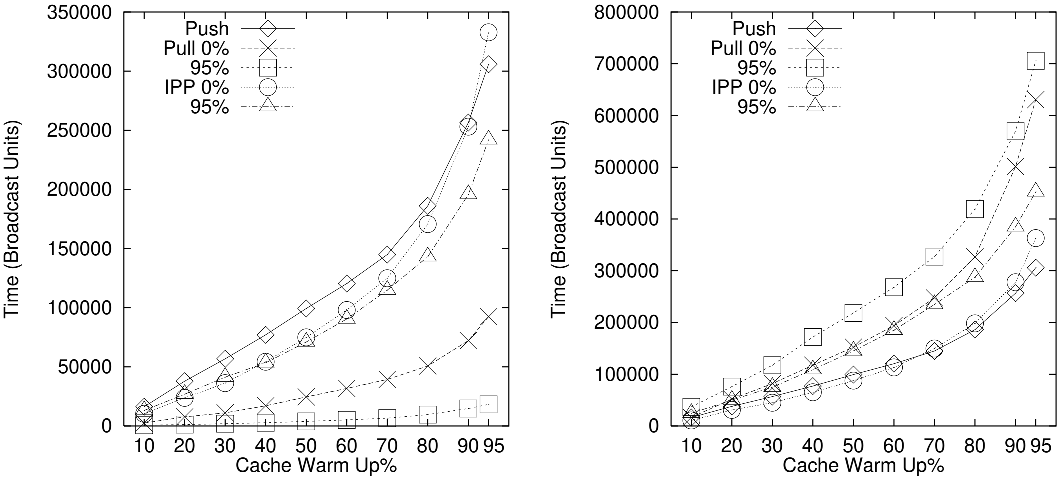
In the preceding graph, one factor driving the relative performance of the $\mathcal{IPP}$ approach compared to the two "pure" approaches was the *PullBW*, that is, the maximum percentage of the bandwidth allocated for servicing pull requests. In theory, this parameter allows $\mathcal{IPP}$ to vary between the performance of *Pure-Push* (*PullBW* = 0%) and *Pure-Pull* (*PullBW* = 100%). Figure 8.1(b) shows the steady-state performance of the $\mathcal{IPP}$ approach (with *SteadyStatePerc* = 95%) for three values of *PullBW*: 10, 30, and 50% (i.e. the same case as in Figure 8.1(a)). The curves for *Pure-Push* and *Pure-Pull* are also shown for reference.

As expected, the performance of $\mathcal{IPP}$ tends towards that of *Pure-Pull* as the bandwidth dedicated to pulled pages is increased. As such, it performs well at lower loads and poorly at higher loads, tending towards the S-curve shape characteristic of *Pure-Pull*. Likewise, with less bandwidth dedicated to pulled pages, the performance of $\mathcal{IPP}$ flattens out, similarly to *Pure-Push*. It is interesting to note, however, that with *PullBW* = 10%, $\mathcal{IPP}$ performs slightly worse than *Pure-Push* even at low system loads. The reason for this behavior is the following: In this case, 10% of the bandwidth is taken away from the pushed pages, resulting in a slowdown in the arrival of those pages (i.e., the Broadcast Disks take longer to rotate). The small amount of bandwidth that is freed up for pull is insufficient to handle the volume of pull requests, even for small client populations. For example, at *ThinkTimeRatio* = 10, 58% of the pull requests are dropped by the server. These dropped requests are likely to be ultimately satisfied by the pushed pages, whose arrival rates have been slowed down by 10%.

Warm-Up Performance

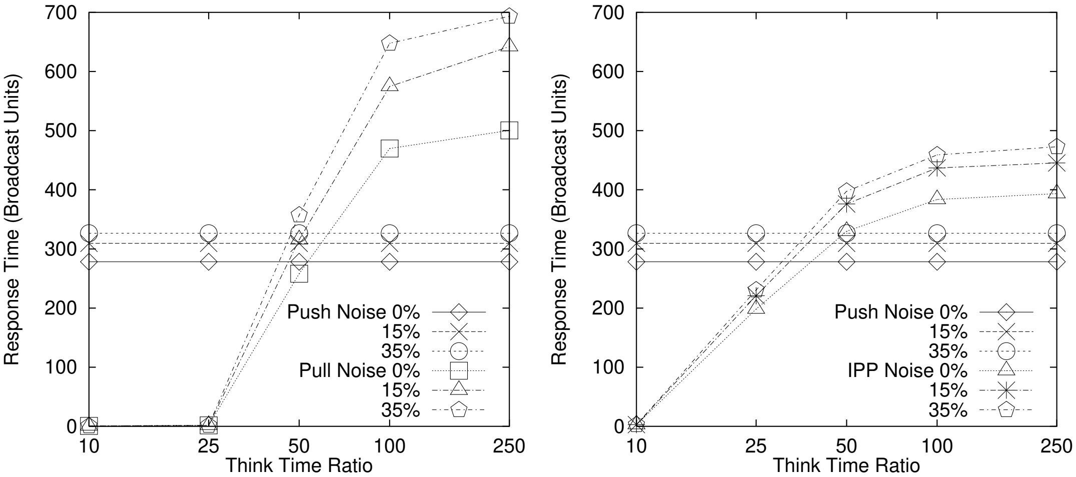
In the first part of this experiment, we investigated the steady-state performance of the $\mathcal{MC}$ under the various approaches. In this section we address the impact of the three approaches on how long it takes a client that joins the system to warm-up its empty cache. This metric is particularly important for applications in which clients are not expected to monitor the broadcast continuously. For example, in an Advanced Traveler Information System [SL96], motorists join the "system" when they drive within range of the information broadcast.

Figure 8.1(a) showed that when the $\mathcal{MC}$ is in steady-state, it performs better for both pull-based approaches when most of the other clients are in steady-state as well. Figures 8.2(a) and 8.2(b) show the time it takes for the $\mathcal{MC}$'s cache to warm up for *ThinkTimeRatio* 25 and 250, respectively, for the same settings shown in Figure 8.1(a) (i.e., *PullBW* = 50%). Along the x-axis, we have the percentage of the *CacheSize* highest valued pages that are in the cache; as the client warms up, this number approaches 100%. Recall that *ThinkTimeRatio*=25 represents a lightly loaded system while *ThinkTimeRatio*=250 corresponds to a heavily loaded system. As can be seen in Figure 8.2(a), under low-moderate loads, *Pure-Pull* allows the fastest warm up, with the other approaches doing significantly worse. For the higher load case shown in



(a) *ThinkTimeRatio =25* (b) *ThinkTimeRatio = 250*

Figure 8.2: Client Cache Warm Up Time, $\mathcal{IPP}$ *PullBW* = 50%



(a) *Pure-Pull* and *Pure-Push* (b) $\mathcal{IPP}$ and *Pure-Push*

Figure 8.3: Noise Sensitivity, $\mathcal{IPP}$ *PullBW* = 50%

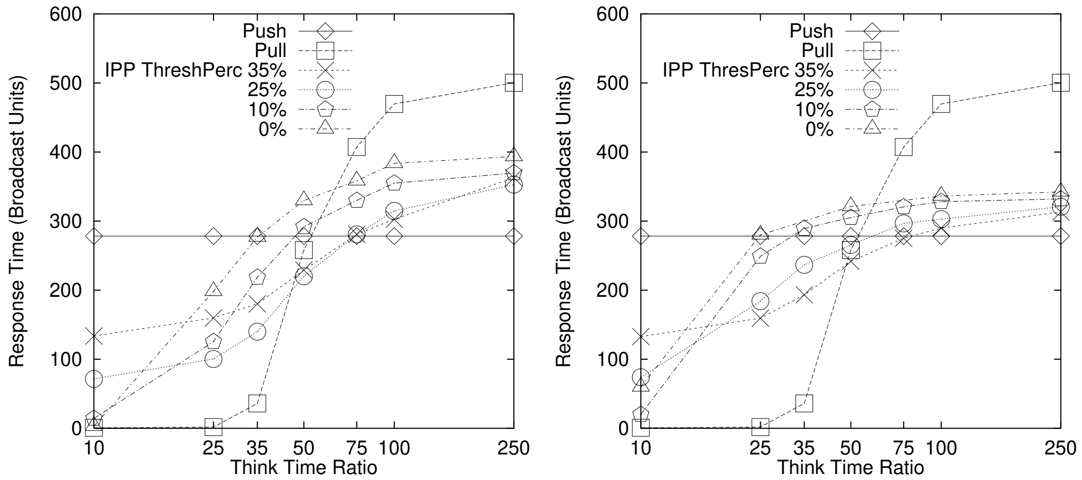
Figure 8.2(b), however, the approaches invert, with *Pure-Push* showing the best warm up performance because the push system delivers data items faster since client requests are dropped due to server saturation. As seen previously, under a heavy load, the $\mathcal{MC}$'s performance for the pull-based approaches is better when the other clients are in a similar state (i.e., warming up, in this case). Note that for the remainder of this chapter, all performance results are shown for the steady-state case (i.e., *SteadyStatePerc*=95%).

Impact of Access Pattern Differences (i.e., Noise)

Both the steady-state and warm-up results showed the negative effect on performance that arises when a client's data access pattern differs significantly from the aggregate access pattern of the other clients in the system. In particular, the results showed that *Pure-Pull* was particularly sensitive to such disagreement when the system was heavily loaded. In this section, we examine this issue further, using the *Noise* parameter which is used to introduce disagreement, as described in Section 4.2. Recall that in this study, *Noise* specifies the degree to which the $\mathcal{MC}$'s access pattern is permuted with respect to the access pattern of the $\mathcal{VC}$ (which is used to generate the broadcast program). A higher *Noise* value indicates a higher level of disagreement between $\mathcal{MC}$ and the $\mathcal{VC}$. Although we report performance results only for the $\mathcal{MC}$, these results are indicative of what *any* of the clients would experience if they differed from the average access patterns of the other clients.

Figures 8.3(a) and 8.3(b) show the performance of *Pure-Pull* and $\mathcal{IPP}$ respectively, for three different values of *Noise*. Both figures also show the performance of *Pure-Push* for those *Noise* values. Turning to Figure 8.3(a), it can be seen that at low system loads, *Pure-Pull* is insensitive to *Noise* but that at high system loads, *Noise* has a substantial negative impact. As long as the server is able to satisfy all of $\mathcal{MC}$'s pull requests quickly, its performance is independent of the access patterns of the other clients. When the server becomes loaded and drops $\mathcal{MC}$ requests, however, $\mathcal{MC}$ becomes dependent on the requests of the other clients. With higher *Noise* values, the other clients are less likely to ask for a page on which $\mathcal{MC}$ might be blocked.

In contrast, Figure 8.3(b) shows that the negative impact of *Noise* becomes apparent at a lower *ThinkTimeRatio* value for $\mathcal{IPP}$ than for *Pure-Pull* (e.g., 25 instead of 50). Here, the server saturates earlier for $\mathcal{IPP}$ because half of the broadcast bandwidth is dedicated to pushed pages. At higher loads, however, $\mathcal{IPP}$, is much less sensitive to *Noise* than *Pure-Pull*. At higher loads, the pushed pages act as a "safety net", that bounds the amount of time that a client will have to wait for a page if one of its requests is dropped by the server. Also, as *Noise* increases, $\mathcal{IPP}$'s performance degrades since its access pattern deviates from the push program broadcast by the server.



(a) *PullBW = 50%* (b) *PullBW = 30%*

Figure 8.4: Influence of Threshold on Response Time

8.3.3 Experiment 2 - Reducing Backchannel Usage

The previous results demonstrated a fundamental tradeoff of the asymmetric broadcast environment, namely, that under very light load, pull can provide far more responsive access than push, but that pull suffers from scalability problems as the load on the system is increased. Beyond a certain *ThinkTimeRatio*, *Pure-Pull* was seen to perform significantly worse than *Pure-Push*. The $\mathcal{IPP}$ approach is an attempt at a compromise between the two "pure" approaches. As such, $\mathcal{IPP}$ was found to perform worse than *Pure-Pull* under light to moderate loads, but better than *Pure-Pull* under heavy loads. $\mathcal{IPP}$ pays a price when the load is low to moderate because it dedicates some of the broadcast bandwidth to push. As the server becomes saturated with backchannel requests, however, it begins to drop the client requests, and the pushed data provides a much needed safety net that ensures that clients eventually receive their data.

$\mathcal{IPP}$ loses to *Pure-Pull* under moderate loads because it sends the same number of requests to the server as *Pure-Pull*, but has less bandwidth with which to satisfy those requests. Thus, the server becomes saturated under $\mathcal{IPP}$ before it becomes saturated under *Pure-Pull*. For example, in the experiment shown in Figure 8.1(a), at a *ThinkTimeRatio* of 50, the server drops 68.8% of the pull requests it receives when $\mathcal{IPP}$ is used, as opposed to only 39.9% of the requests when *Pure-Pull* is used.

One way to reduce the number of pull requests sent to the server under the $\mathcal{IPP}$ approach is to constrain the pages which clients are allowed to ask to be only those that are "farthest away". As described in Section 8.2.1, we use the notion of *threshold* to accomplish this. $\mathcal{IPP}$

with a threshold works as follows: On a cache miss, clients check the push program to determine when the missed page is scheduled to appear next on the broadcast.[7] If the page is scheduled to appear within the threshold, then the client must wait for the page without issuing a pull request. Thresholds are expressed as a percentage of the push period (i.e. the major cycle length of the Broadcast Disk).

In the results shown so far, $\mathcal{IPP}$ used a threshold (called *ThresPerc*) of 0% — any cache miss would result in a pull request being sent to the server. In contrast, a *ThresPerc* of 25% would restrict the client from sending pull requests for *any* pages that are scheduled to appear within the next quarter of the push period (i.e., the major cycle). The threshold, therefore, provides a knob that trades-off expected delay versus backchannel usage and server congestion.

The intuition behind threshold is to save the use of the backchannel for those pages that currently would induce the longest response time penalty. Threshold is a simple mechanism for improving the usefulness of the backchannel. It can be applied at individual clients, and requires only that they be aware of the broadcast schedule for the pushed data.

Figure 8.4(a) shows the performance for various values of *ThresPerc* when the *PullBW* is set at 50% (as in the previous experiment). Also shown for reference are *Pure-Push* and *Pure-Pull*. The curve for a *ThresPerc* of 0% is identical to that for $\mathcal{IPP}$ in Figure 8.1(a). With no threshold, $\mathcal{IPP}$ becomes slower than *Pure-Push* when the *ThinkTimeRatio* exceeds 35. In contrast, with a *ThresPerc* of 25%, $\mathcal{IPP}$ requests only half of its missed pages from the server[8] and therefore, it remains better than *Pure-Push* up to a *ThinkTimeRatio* of 75 — roughly a factor of two improvement in the number of clients that can be supported.

Under low loads (e.g., *ThinkTimeRatio*= 10), threshold hurts performance by unnecessarily constraining clients — there is plenty of server bandwidth to handle all the requests of the small client population. As clients are added however, the threshold begins to help performance. In this case, however, it is interesting to note that a threshold of 35% performs worse than one of 25% up to a *ThinkTimeRatio* of 50. Prior to that point, the threshold of 25% is sufficient to offload the server (e.g., no requests are dropped) so the additional threshold simply makes clients wait longer than necessary for some pages.

Figure 8.4(b) shows the performance for the same settings except with the *PullBW* reduced to 30%. In this case, the reduced bandwidth available for pull slots causes the server to become saturated earlier (e.g., for *ThresPerc* = 25% and *ThinkTimeRatio* = 25, the server drops 9.4% of the pull requests). As a result, a *ThresPerc* of 35% provides the best performance for $\mathcal{IPP}$ in all cases here except under very light load. This also translates to better scalability: $\mathcal{IPP}$ crosses *Pure-Push* at *ThinkTimeRatio* = 25 with no threshold but at *ThinkTimeRatio* = 75 with a threshold of 35%. This translates to roughly a factor of three improvement in the number of clients that

[7]Since the client does not know what the server plans to place in the pull slots, the time-to-next-push is an upper bound on how long the client will wait for the page.

[8]Due to the shape of the multi-disk used to structure the push schedule, approximately half of the pages appear at least once in the first 25% of the broadcast period here.

can be supported before losing to *Pure-Push*.

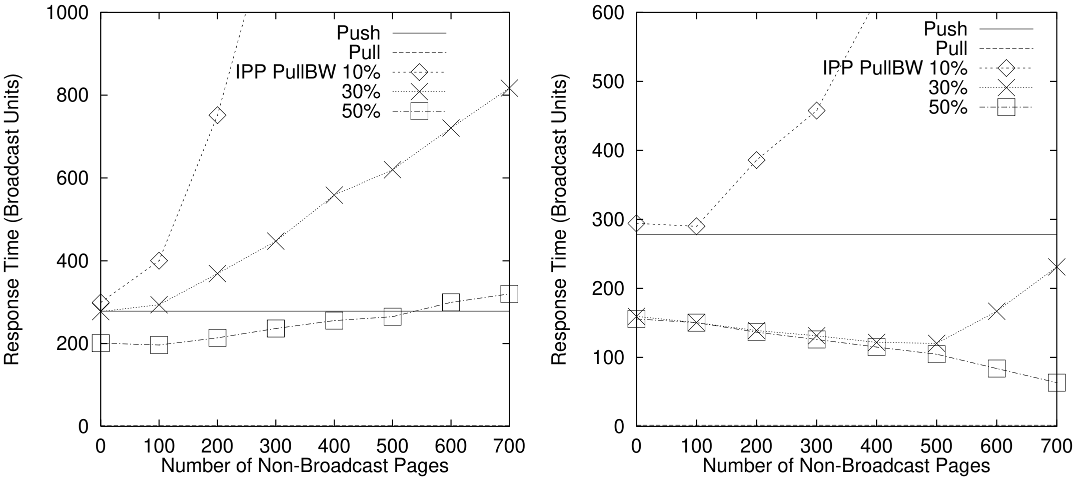
As the above graphs show, $\mathcal{IPP}$ with different threshold values never beats the best performance of *Pure-Pull* and *Pure-Push*. However, as pointed out earlier, the two "pure" algorithms can degrade in performance rapidly if the system moves out of their niche server load range. $\mathcal{IPP}$, on the other hand, while never having the best performance numbers, with an appropriately chosen value of threshold, can provide reasonably good performance over the complete range of system loads.

8.3.4 Experiment 3 - Restricting the Use of Push

An important issue that has arisen in the preceding experiments is the impact of the allocation of broadcast bandwidth to pushed pages and to pulled pages. As discussed in Section 8.3.2, one way to adjust this allocation is to change the *PullBW*. Up to this point, however, our approach placed *all* pages on the push schedule. The Broadcast Disks model provides a flexible way to do this, as it allows the proportion of bandwidth given to particular groups of items to be assigned in accordance with their popularity. Changing the *PullBW* in the previous experiments, therefore, affected the period length of the push schedule, but did not impact its contents or the relative frequencies at which pages appear in the schedule.

An alternative (or actually, complementary) approach to changing the *PullBW* is to restrict the set of pages that are placed in the push schedule. That is, rather than pushing the entire database, choose a subset of the pages to push, and allow the rest to be obtained by pull only. The motivation for doing so is as follows. Being a broadcast channel, every page transmitted by the server is delivered to all the clients. Thus, only the "globally popular" pages should be broadcast as part of the push schedule; clients should pull the other pages which are "locally popular" via the backchannel. A similar approach was also advocated in [IVB94b, VI94]. Although that work was based on a somewhat different system model and had a different objective function, the tradeoffs that arise in both environments are similar. In general, as discussed above, placing all items on the push schedule provides a "safety net", which bounds the time for which a client will have to wait for a data item. Such a bound is particularly important in our environment, in which the server responds to heavy load by dropping requests. The downside is that placing all items on the push schedule can result in a very long broadcast cycle and consequently, a significant waste of broadcast bandwidth. Items that are of limited popularity take up broadcast slots that could be used for pushing more popular pages, or for satisfying more pull requests. The Broadcast Disks paradigm mitigates this problem to some extent, by providing the flexibility to place unpopular pages on slowly spinning "disks". The fact remains, however, that pushing pages onto the broadcast is wasteful if those pages are unlikely to be accessed by the majority of the population.

Figure 8.5 shows the performance of the $\mathcal{IPP}$ approach for two different values of *ThresPerc* and varying *PullBW* settings as pages are incrementally truncated from the push schedule.



(a) *ThresPerc* = 0% (b) *ThresPerc* = 35%

Figure 8.5: Restricting Push Contents, *ThinkTimeRatio* = 25

The push program is made smaller by first chopping pages from the third (slowest) disk until it is completely eliminated and then dropping pages from the second (intermediate) disk. Figures 8.5(a) and 8.5(b) show the case when $\mathcal{IPP}$ does not use a threshold (*ThresPerc* = 0%) and when using a *ThresPerc* of 35% respectively for a lightly loaded (*ThinkTimeRatio* = 25) system. At the left of the x-axis, the push schedule is fully intact, while at the far right, the third disk is removed and 200 pages are dropped from the second disk. Recall that the third disk has a size of 500 pages. The main point that these figures show is that if pages are to be removed from the broadcast, then adequate pull bandwidth must be provided to allow clients to obtain those pages.

The performance of *Pure-Pull* and *Pure-Push* are independent of the number of pages not on the push schedule and thus, are straight lines with response times of 2 and 278 units respectively.[9] In all the following figures, since the response time of $\mathcal{IPP}$ for *PullBW*=10% rises dramatically, the y-axis is clipped to keep the graphs in scale. In Figure 8.5(a), the clients request every miss for $\mathcal{IPP}$ since there is no threshold. The client miss rate increases as more pages are dropped from the push schedule, causing the server to saturate and start ignoring requests. Consequently, the response time can degrade rapidly if the pull bandwidth does not keep up with the demand (e.g., *PullBW*=10%). Note that unlike the case in which the whole database is broadcast, there is no upper bound on the latency of a page not on the push schedule. *PullBW*=50% does somewhat better than *Pure-Push* until the entire third disk is dropped (500 pages).

Figure 8.5(b) shows the performance for the same simulation settings as Figure 8.5(a), except that *ThresPerc* is set to 35%. Since clients filter some of their requests due to threshold,

[9]The line for *Pure-Pull* is not visible in Figures 8.5(a) and 8.5(b) because of overlap with the x-axis.

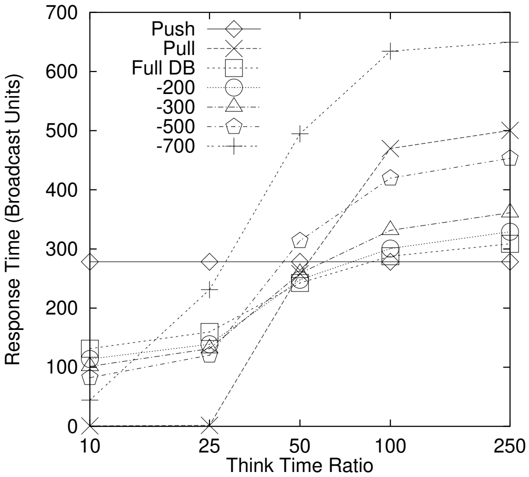


Figure 8.6: Server Load Sensitivity for Restricted Push

PullBW = 30%, *ThresPerc* = 35%

the server is not as heavily loaded as before and this shows in the much slower performance degradation for *PullBW* of 10%. Both the other *PullBW* algorithms outperform *Pure-Push* here since the system is lightly loaded. Consequently, dropping more pages can significantly improve performance (up to a point) since there is sufficient pull bandwidth to satisfy requests. For example, for *PullBW* of 50%, the response time numbers are 155 and 63 *broadcast units* at the left and right end of the graph respectively. However, beyond 500 dropped pages, *PullBW*=30% starts to degrade. This is because the 30% bandwidth for pull requests is no longer sufficient to handle the additional misses from the second disk pages which are no longer on the push schedule.

Note that the penalty incurred by $\mathcal{IPP}$ for dropping a request for a page not on the broadcast schedule is not only much higher than for *Pure-Push* (due to the absence of the "safety net") but also much higher than *Pure-Pull* since the push schedule uses a portion of the bandwidth. Thus, unlike in Experiment 1 and Experiment 2, the response time is *not bounded* in the worst case by *Pure-Pull*. This is seen in Figure 8.6 which shows the simulation space from a viewpoint similar to the earlier experiments with the server load (as determined by *ThinkTimeRatio*) on the x-axis and the client response time on the y-axis. The graph shown is for a *PullBW* of 30% and *ThresPerc* of 35% and the legend for $\mathcal{IPP}$ represents the size of pages chopped from the push schedule. As is seen, when 700 pages are dropped, $\mathcal{IPP}$ does worse than *Pure-Pull* all through the range of the x-axis. The lesson from the graphs in Figure 8.5 of the need for adequate pull bandwidth to respond to client requests is again apparent here in the crossover of the lines between *ThinkTimeRatio* of 25 and 50. If the system is underutilized, chopping more pages makes sense since there is enough pull bandwidth to serve client requests. Once the system saturates (beyond *ThinkTimeRatio* of 25), the performance hit due to dropped requests begins to show. The inversion of the performance order of the $\mathcal{IPP}$ algorithms from the lightly loaded left-hand side to the saturated right side of the graphs is due to these dropping of requests and the upper bound in latency provided by the push-schedule as fewer pages are chopped from it.

As in the previous two experiments, while a restricted push schedule cannot be expected to beat the best performance numbers of *Pure-Push* and *Pure-Pull*, the above results show that the $\mathcal{IPP}$ algorithm with an appropriately chosen subset to be broadcast (along with proper values of threshold and pull bandwidth), can provide consistently good performance over a wider range of server load than the two "pure" algorithms.

8.3.5 Summary of Results

We have seen that when there is little or no contention at the server, pulling pages using the backchannel with no server push (*Pure-Pull*) is the best strategy. If the server is lightly loaded, all requests get queued and are serviced much faster than the average latency of pages on the Broadcast Disk. At the other extreme (typically at the right-hand side of the response time graphs), the server is saturated and the probability of a backchannel request being dropped is extremely high. Consequently, in this region, the best strategy is to use *Pure-Push* since it provides a "safety net" and puts an upper bound on the delay for any page. The asymmetry is extreme enough to make *Pure-Pull* ineffective justifying the intuition that push-based techniques are best suited for highly asymmetric settings.

Use of either of the two "pure" algorithms in a system with widely varying loads can lead to significant degradation of performance since they fail to scale once the load moves away from their niche domain. Consequently, we introduced the $\mathcal{IPP}$ algorithm, which combined *Pure-Pull* and *Pure-Push* in a straightforward manner to provide consistent performance over the entire spectrum of system load. Since $\mathcal{IPP}$ suffers the same bottleneck problems of any pull based approach, we studied three techniques to improve its scalability and performance by minimizing the time spent in saturation.

The first approach is to adjust the pull bandwidth. Increasing it might move the server away from saturation since the rate at which the queue is serviced also increases. However, the downside of increasing pull bandwidth is the reduction in the push bandwidth, slowing the broadcast delivery and introducing a fundamental tradeoff. The limiting cases of this are *Pure-Pull* (*PullBW*=100%) and *Pure-Push* (*PullBW*=0%). An interesting observation about pull bandwidth is that while lower values of *PullBW* (e.g., 30%) do not produce the best results when the system is lightly loaded, neither do they produce the worst results when the system is heavily loaded and thus, are suited for systems with dynamic loads.

Another way to delay the saturation is the use of a threshold. Instead of flooding the server with every possible miss, clients use discretion by only requesting their most "expensive" ones. This has the effect of delaying the point at which the server queue saturates.

The final technique to avoid saturation is to successively chop larger pieces from the slowest part of the broadcast schedule to increase the available bandwidth for pull. Since *PullBW* essentially predetermines the split between pull and push bandwidth allocations, we must increase *PullBW* in order to realize a benefit from this approach. This technique has the disadvantage that if there is not enough bandwidth dedicated to pulled pages, performance can degrade severely since clients will not be able to pull non-broadcast pages and there is no upper-bound on the latency for those pages provided by the push schedule. Using a threshold with the "chopped disk" can help considerably since all non-broadcast pages pass the threshold filter and the effect is to reserve more of the backchannel capability for those pages.

We also studied the effect of differences in access patterns in studies of warm-up times and noise. For warm-up, in a lightly loaded system, *Pure-Pull* warms up fastest, while in a heavily loaded system, *Pure-Push* does best. In general, a client will warm up quicker if other clients are in a similar state. Disagreement in access patterns is represented by *Noise*. We showed that for a lightly loaded system, *Noise* has little impact since clients can pull what they need; however, for a heavily loaded system, *Noise* has a large negative impact. $\mathcal{IPP}$ saturates earlier, but is overall less sensitive to noise because of the "safety net" effect of the push program.

8.4 Related Work

While dissemination-based environments have been studied for a while now, investigating the combination of push and pull to simultaneously deliver data has only recently started to receive attention. [IVB94b, VI94] studied the issues in a mobile framework and is closely related to the results in this chapter, in terms of studying the trade-offs between pull and push in a broadcast environment. They list a number of possible intermediate data delivery options between the two extremes of pure-pull and pure-push. They propose an algorithm to split the database into a "publication" group and an "on-demand" group in order to minimize the number of uplink requests while constraining the response time to be below a predefined upper limit. Though similar in spirit to Experiment 3 (Section 8.3.4), the goal of our work is different in that we try to find the best response time for the client, while budgeting the use of the backchannel. However, the biggest difference is in the model of the server. They use an M/M/1 queuing model for the backchannel which when unstable, allows a queue of infinite length. Our environment is not accurately captured by an M/M/1 queue. Requests and service times are not memoryless, due to caching and the interleaving of push and pull slots dictated by *PullBW*. Also, the server in our model uses a bounded queue, which drops requests if it becomes full. Another difference between the models is that in our environment, any page can be pulled through the backchannel as long as its delay is greater than the threshold. In [IVB94b, VI94], only pages in the "on-demand" group can be requested over the backchannel. They also assume an implicit symmetric model where both the back and the front channel share a common medium. Thus, while there is a correlation between their results and ours when their model is stable (corresponding to a low *ThinkTimeRatio* in this work), in general, those results are not directly applicable here. A similar model was used is earlier work on Videotext and Teletext systems [AW85, Won88]. [Won88] presents a survey of some of the analytical studies on one-way (push only) [AW85, AW87], two-way (pull only) [WA85] and hybrid broadcast (both push and pull) [WD88] in this framework. In [WA85], they address a request/response pull-only model in the Videotext system. As in our framework, the response is broadcast to all the clients. The problem of server saturation demonstrated in the graphs in this chapter is also shown analytically there. In [WD88], that work is extended to include delivery via both a cyclic push and a pull mechanism. Similar to [IVB94b], in [WD88] the data is classified into two distinct groups each serviced either via cyclic push or via pull. In this work we have assumed a FCFS model for the backchannel queue. [WD88] proposes a number of other potential alternatives including *most-requests-first*, which broadcasts the page with the most pending requests and *longest-wait-first*, which sends out the page whose request has accrued the greatest delay.

More recently, [SRB96, SRB97] have proposed an alternative hybrid push and pull model to deliver data. Unlike the model in this thesis and the papers listed above, in this approach, the responses to the pull requests are unicast only to the client which made the request. Also, they use an adaptive strategy based on the request pattern visible to the server to dynamically change the subset of the data which is serviced via cyclic push and that is delivered via point-to-point pull response.

8.5 Conclusions

In this paper, we addressed the problem of adding a backchannel to a Broadcast Disks environment. Introduction of a backchannel, allows the clients to play a more active role in accessing data by making pull requests to supplement the regular delivery via the push schedule. However, use of the backchannel to pull data creates two fundamental problems in a broadcast-based environment. One, the need to respond to pull requests forces the server to divide the downstream bandwidth between the regular push schedule and responses to the pull requests and this creates an interesting tradeoff. The second issue is a problem that afflicts all pull-based systems. As the number of client increase, pull-based systems fail to scale in performance due to contention at the server and the backchannel.

Due to the above problems, in any environment which support data delivery via both push and pull, there is a continual tension between balancing the latency of the push mechanism and the latency of the pull requests. Since the downstream bandwidth is fixed, trying to improve one necessarily leads to degrading the other. Thus, the role of both the server and the clients in such an environment is to cooperate to find an appropriate balance so as to maximize the client performance. In this chapter, we studied a number of server-side and client-side mechanisms to achieve that goal.

The experiments demonstrated that when the system is lightly loaded, pull-based schemes are more responsive than push-based schemes. As the contention for the backchannel and the server increases due to increase in the number of clients, push-based schemes outperform pull-based schemes because they provide a "safety net" by limiting the maximum wait for a page. The $\mathcal{IPP}$ algorithm was introduced which combined both push and pull and attempted to mitigate their individual drawbacks. While $\mathcal{IPP}$ never achieved the best performance of either pure-pull and pure-push, it was found to be much more robust in performance over a wider spectrum of system loads. Also, a number of techniques to fine tune $\mathcal{IPP}$ were investigated. In particular, we demonstrated how using a threshold, adjusting the pull bandwidth, and reducing the size of the scheduled push program can all contribute to improved performance and scalability over a wide range of workloads.

The contribution of this chapter is to delineate the issues and the tradeoffs that one must confront in this type of environment. As the experiments demonstrated, it is not trivial to find a proper balance between push and pull. However, a scheme like $\mathcal{IPP}$ which can juggle these conflicting demands is the one that achieves good performance and scalability.

Chapter 9

Designing a Broadcast Disks Prototype

9.1 Introduction

This chapter describes the design and implementation of a Broadcast Disks prototype. The prototype runs on a cluster of Intel Pentium-based computers running Windows NT which are networked using 10/100 Megabit/sec ethernet. The results in this chapter were generated using a configuration of one server and up to three clients. However, given the inherent broadcast nature of the ethernet, the number of clients can scale to as many as can be supported on an ethernet. IP Multicast [Com95] was used as the basis for data delivery from the server to the clients. The prototype was written in C++ and it contains about 15,000 lines of code.

The motivation for the prototype was three-fold. They were:

- *Feasibility of a push-based system.* The primary goal of building the prototype was to prove data dissemination as a viable alternative. There have been earlier efforts at building push-based systems (e.g., Datacycle [BGH$^+$92]). However, unlike these earlier projects, the goal of the prototype was to use off-the-shelf hardware and standard software components. By implementing a working prototype, we hoped to also gain valuable experience about the tools available and the current state of the technology for building such systems.

- *Validating the correctness of the simulation studies.* To make the simulation feasible, a number of assumptions were made about the broadcast environment. Another goal was to justify these assumptions and determine the validity of the results presented in the previous chapters.

- *Analyzing issues not addressed in simulations.* The simulation model deliberately ignored a number of features of the broadcast environment because a) it simplified the design and b) given the novel nature of this environment, we lacked data and experience to make an accurate assessment of these factors. Consequently, the simulation results are valid only for the space where these factors have no influence on system performance. The goal of the prototype was to broaden the scope by also including these factors in the performance study. In particular, we were interested in understanding the impact of continually sampling the broadcast at the client and the role of the server cache on the overall performance.

As in the simulator, the two main design components were the client and the server. In this chapter, we only provide the outline of the various design components and justify our choices. For the details of the various subcomponents, the reader is referred to [Bos97]. Also, we only present a subset of the results generated using the prototype. The complete set of these and other experiments are available in [Cui97].

The outline of the rest of this chapter is as follows. In the next section, we highlight some of the major differences between the simulator and the prototype. Then, we briefly describe the design of the prototype server and the client. In Section 9.4, we present performance studies using the prototype. Finally, we summarize the chapter.

9.2 Key Differences with the Simulator

The architecture of the prototype borrows significantly from that of the simulator. However, there are significant differences between the two designs. They are as follows:

- *Server caching and disk storage.* The design of the server was a significantly more challenging task in the prototype. In the simulator, the sole responsibility of the server module was to ensure that pages were transmitted in accordance to the Broadcast Disks schedule. Since the time was measured in *logical* units, which counted the *number* of pages and not the actual time, there were no *real-time* constraints on the data transmission. On the other hand, in the prototype, the data is stored on the disk and it is transmitted via the network. Consequently, the server has to move the data from the disk to the network through the various system layers in a time efficient manner and the (server) cache was used as a means to facilitate this process. To ensure that the pages are broadcast at regular intervals (and not in a bursty fashion), the server required that the data be cache-resident before it was due for transmission. Consequently, cache management at the server, absent in the simulation model, is a crucial issue for the prototype.

- *The network.* In the simulations, the network was modeled as a generic FIFO queue. In the prototype, an ethernet network was used to connect the clients and the server.

Consequently, the clients and the server required network-level modules to co-ordinate the data transfer. One of the design goals was to isolate the upper layers from the lower network layer by providing a high-level interface for data transfer and thus, increase portability.

- *Distinct client and server machines.* In the absence of an actual physical network in the simulation studies, the client and the server process could reside on the same machine. In the prototype, the clients and the server are on physically different machines. Consequently, information which could be exchanged between the client and the server via shared data structures in the simulations have to be explicitly transferred over the network in the prototype. This entails augmenting the broadcast program to carry the additional *meta-information* (e.g., time to next arrival for each page). Also, the prototype design requires additional software modules to package and extract such information at the server and the client respectively.

These design differences above also have an important implication on the client performance:

- *Real-time constraints at the client.* In the simulations, the time was measured in logical *Broadcast Units* which counted the *number* of pages and not the actual time to transmit them. Consequently, in the absence of any timing constraints, the server broadcast pages at a rate requested by the clients. In effect, the rate at which the client consumed a page, controlled the rate at which the server broadcast them. In a real system, since the server has to support multiple clients, it cannot be expected to be aware of each client's processing capabilities. Consequently, the client application has to consume each page before the next one arrives; else it risks failing to process all the data. Moreover, there is additional overhead at the client of managing the caching data structures and the operating system overhead of traversing the network stack. A slow client, thus, has a potential to "drop" pages from the broadcast due to overflowing of the buffers on its network interface. This is one of the key differences between the two setups and its effect on real-time performance will be studied in greater detail in Section 9.4.

In the next section, we highlight the server design followed by a description of the prototype client.

9.3 Design Description

The prototype consists of a page-server which provides a GUI-based interface to control the various server setup parameters (e.g., page size, Broadcast Disks inputs etc.) and one or more clients which access the broadcast. The clients similarly have an interface which allows the user to specify the different client-side parameters (e.g., the cache management algorithm). The design allows for both push-only delivery from the server to the clients and duplex communication where the clients can make explicit pull requests.

9.3.1 Server Design

As in any client-server system, the primary responsibility of the server is to store the data and provide efficient access to it. Figure 9.1 shows the different components of the server. In a dissemination-based Broadcast Disks environment, the server also has to ensure that the data is pushed to the clients in a regular and timely fashion. To achieve this, the data has to be moved efficiently from the disk, through the cache and onto the network. As pointed out earlier, this was not an issue in the simulator since the database was assumed to be cache-resident there.

In this subsection, we will briefly highlight the responsibilities of the six major subcomponents that constitute the server. The internal architecture of the components are described in more detail in [Bos97]. Figure 9.2 shows the structure of a "page" which is delivered from the server to the client. It consists of the raw data and additional headers which are added by the various components. Example fields in each of the headers is shown on the right.

Broadcast Policy

This is the control center of the server. It is responsible for determining the next page going out on the network. In the Broadcast Disks environment, there are three categories of pages:

- *Push Pages.* These are pages which are part of the regular Broadcast Disks program.
- *Pull Pages.* These are responses to explicit backchannel requests from the clients. Analogous to the *PullBW* parameter in the simulator, the server has a parameter which determines the maximum amount of bandwidth which should be allocated to such pages.
- *Propagation Pages.* This is an issue only in the presence of updates and if the server follows a propagation-based scheme[1]. In this case, the server periodically interrupts the regular broadcast to send out a propagation list containing new values of pages updated earlier. The granularity of the propagation period and the pages which constitute this list is determined by the nature of the propagation-based scheme (e.g., minor or major cycle propagation). This can be input as a server startup parameter.

[1]Updates in the system may also require the server to send out invalidations. Since these messages are small, space is allocated in the header of every page for a short invalidation list. The details of the format of each page and this list is beyond the scope of this description.

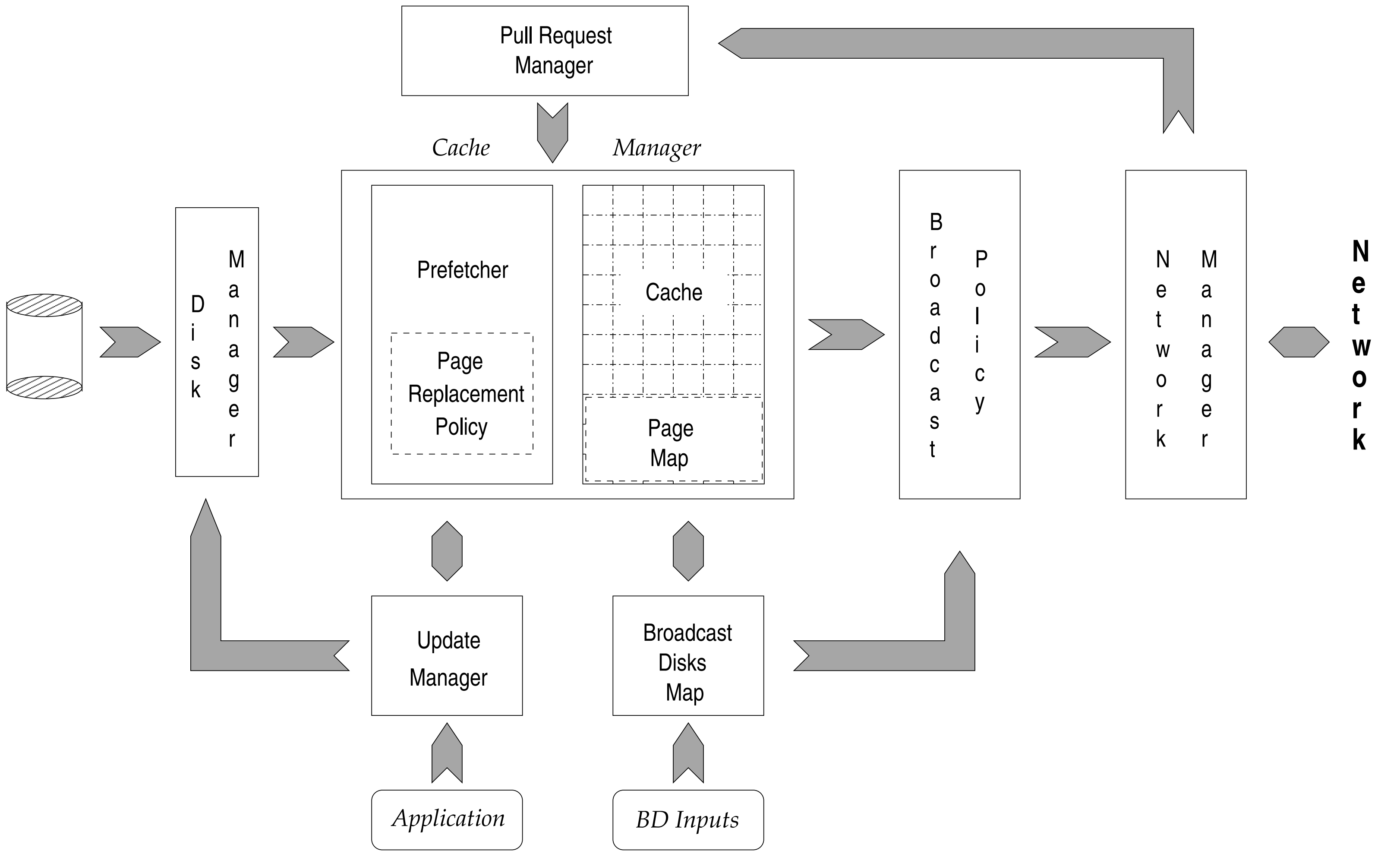


Figure 9.1: Server Block Diagram

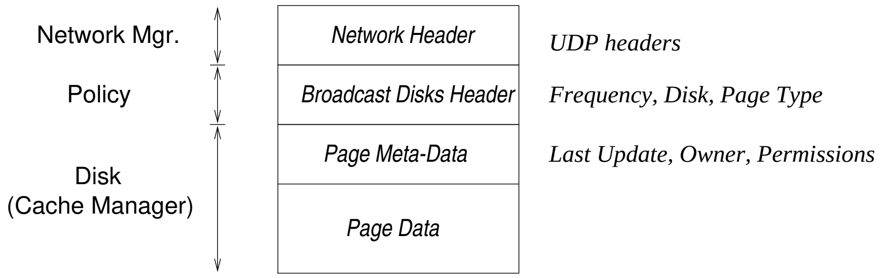


Figure 9.2: Broadcast Page Structure

The *Policy* module runs as a separate thread of control and consists of a three step loop. In the first step, it determines the nature of the page which should be sent out next. In the following step, it requests the cache manager to supply the scheduled push, pull or propagation page depending on the outcome of the previous step. Finally, once it receives the page from the cache manager, it adds a header to the data providing program related information (e.g., its frequency of broadcast, time to next arrival etc.) as shown in Figure 9.2. It then forwards the page to the *Network Manager* (described below) which eventually drops it onto the network.

Cache Manager

The server cache is an intermediate buffer between the server disk, where the data resides and the network which is the final destination of the data. As network speeds have improved, server caches have become critical in offsetting the impedance mismatch between the (lower) data retrieval rates from the disk and (higher) data delivery capacity of the network. In a periodic dissemination-based system, the server has an additional burden of ensuring that the data delivery is regular and pages are not sent out in bursts. Consequently, in order to minimize the variance in transmission times for successive pages, the primary responsibility of the *Cache Manager* is to make sure that the page to be broadcast next is always cache-resident. The Cache Manager consists of two key modules— the *Prefetcher* and the *Cache*.

The *Prefetcher* is responsible for making pages cache-resident before they are due for transmission. It achieves this by staying ahead of the Policy module by a predetermined lookahead. Since the Broadcast Disks program is fixed, the push schedule is available a priori (from the *Broadcast Disks Map* module described next). However, the pull requests and the updates cannot be predicted. Thus, the *Prefetcher* wakes up periodically, queries the three input sources (push, pull and propagation) for their scheduled list of pages and brings them into the cache. The *Prefetcher* module, thus, has two interrelated knobs which can be adjusted — *LeadSize*, which determines the lookahead and *CycleSize* (in pages) which is how long the *Prefetcher* "sleeps" before it is woken up. The lookahead parameter determines the slack available to the *Cache Manager* in case of contention for the disk from other applications. The *CycleSize* parameter implicitly also determines how many pages are prefetched every time the *Prefetcher* is active. This creates an interesting tradeoff. The more often the Prefetcher runs, the greater the overhead. However, the shorter the "sleep" time, the more responsive it is to updates and pull requests.[2]

The *Cache Manager* also has a subcomponent, the *Page Replacement Policy*, to choose victims to make space for the prefetched pages. The *Cache* is the passive component which reacts to requests for pages from the Policy module. Depending on the type of page requested, it contacts the appropriate source (Push, Pull or Update module) for the next scheduled page and responds appropriately. The *Page Map* sub-module is responsible for translating the page identifier to a physical memory address.

Network Manager

The *Network Manager* is the server's interface to the outside world. It hides the low-level network calls by providing an uniform high-level interface to the other modules. This decoupling adds to the software portability allowing the server to either change the network protocol used for data delivery (TCP, UDP etc.) or the type of the network technology (ethernet, token ring etc.) without affecting the other modules. The results in Section 9.4 are for IP multicast (based on UDP) on an ethernet network.

The *Network Manager* has two responsibilities — transmit data on the network and receive backchannel requests from the clients. Once the *Policy* module receives the page to be sent out next, it forwards it to the *Network Manager*. The page is then tagged with additional network protocol specific headers (Figure 9.2) and queued for transmission. To prevent overflow of the server's network interface buffers, the *Network Manager* slows down if there are too many pages waiting to be transmitted. The *Network Manager* module also monitors backchannel requests from the clients. When such a request is made, the message is forwarded to the *Pull Request* module. Unlike the downstream channel, clients connect to the server using a point-to-point TCP connection.

Disk Manager

This module is responsible for the layout of the pages on the server disk and efficient access from it. In the simplest case, the *Disk Manager* uses the standard file system calls to access the data and lets the operating system control the layout. However, the design allows for the module to control the clustering of pages in order to optimize the overall access latency.

The *Disk Manager* is also responsible for updating pages on the disk. Since updates are generated from the *Update Manager* and the reads arise from the *Cache Manager*, it has to ensure

[2]Typically, unless pulls and updates are very rare or the *CycleSize* is extremely long, the periodic polling of the input sources by the *Prefetcher* does not affect the responsiveness. For the experiments presented in this chapter, the *CycleSize* was maintained at 10% of the cache size.

that the reads always see the most recent value of the data.

Pull Request and Update Manager

The *Pull Request* module handles the backchannel requests from the clients. It is very similar to the corresponding module in the simulator. It queues the requests as they arrive and if there is a pre-existing request for a page already, a subsequent request is ignored. It also periodically provides the list of unsatisfied requests when the *Prefetcher* is activated, so that they can be prefetched into the cache. The various pull parameters (e.g., size of queue, pull bandwidth percentage on the downstream channel etc.) are taken as input during startup.

Similarly, the *Update Manager* is responsible for the updates which are posted at the server. Every update generated is forwarded to both the *Cache Manager* and the *Disk Manager* to ensure that the permanent disk copy and any cached copy are updated to reflect the new value.

Broadcast Disks Map

The Broadcast Disks module controls the structure of the push schedule. It takes the user given inputs for the Broadcast Disks program and creates a schedule using the algorithm given in Section 3.2.2. It is also responsible for providing all the Broadcast Disks meta-data such as the frequency and time until next arrival for a page to the *Policy* box.

9.3.2 Client Design

The block diagram for the client is shown in Figure 9.3. Each of the modules at the client have a one-to-one correspondence with a module at the server and perform analogous (or, inverse) functions.

Network Manager

The *Network Manager* is the client's gateway to the Broadcast Disks environment. The role of the corresponding module at the server is to multiplex the data from various sources (push, pull or update) into the network channel. The role of the client *Network Manager* is to do the inverse — it de-multiplexes each page to determine the type and (if necessary) forwards it to the client *Cache Manager*.

In a demand-driven mode of client operation, the client *Cache Manager* (described next) provides the identifier of the page required by the application to the *Network Manager*. When these pages are broadcast, the *Network Manager* forwards them to the *Cache Manager*. However, if the client is prefetching, every page is forwarded.

The *Network Manager* is also responsible for the backchannel communication. On receiving the request from the *Pull Manager*, it sends a UDP message to the server containing the

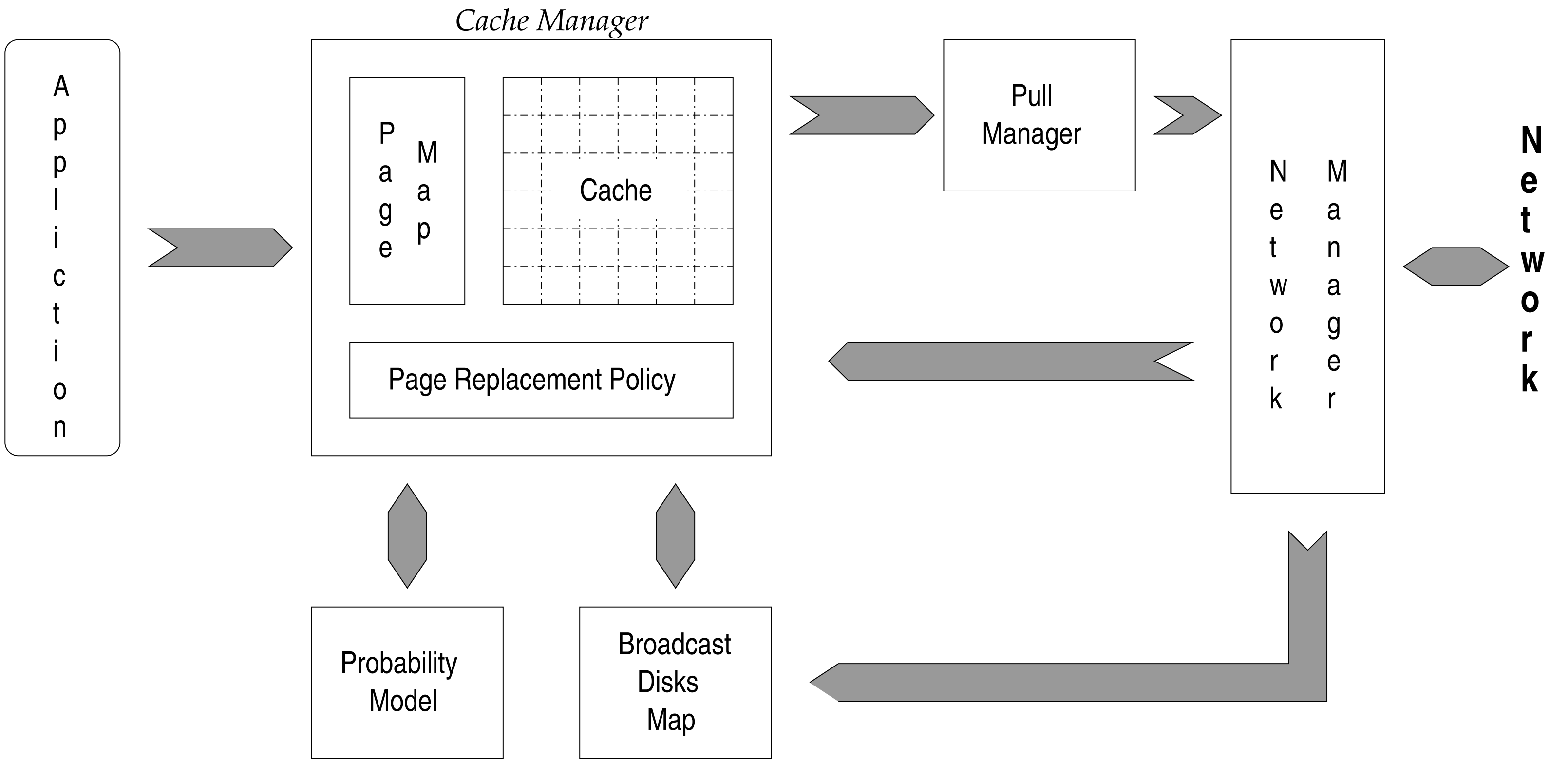


Figure 9.3: Client Block Diagram

identifier of the page.

Cache Manager

The role of the *Cache Manager* at the client is similar to that in the server. It serves as a buffer between the application and the network. As the results in the previous chapters have demonstrated, the cache plays a critical role in the client performance.

The *Cache Manager* consists of the *Cache* and the *Replacement Policy* modules. A modular design allowed us to test different page replacement policies. If the page requested by the application is not available, the cache forwards the request to the *Pull Manager*, which forwards it to the *Network Manager*. When the page arrives on the broadcast, the *Network Manager* delivers the page to the *Cache*. In response, it calls upon the *Replacement Policy* to choose a victim.

Probability Model and Broadcast Disks Map

The *Probability Model* module samples the page requests arriving at the cache and builds a model of client access. This model used by the page replacement algorithm to choose the victim.

The *Broadcast Disks Map* module is analogous to its namesake at the server. It maintains all the meta-information about each page on the broadcast (e.g., frequency of broadcast, the disk it is on, time until next arrival etc.) and additional information about the structure of the broadcast.

Pull Manager

This is the module which handles cache misses at the client and it is responsible for deciding whether a page should be explicitly requested via the backchannel or if the client should wait for the regular broadcast to deliver the data item. In the absence of a backchannel, it forwards the miss request from the *Cache Manager* to the *Network Manager*. However, if there is support for a backchannel, the *Pull Manager* uses a *threshold-based* filter (Section 8.2.1) as in the simulator, to determine if an explicit request should be made. The threshold parameter is taken as an input during startup.

9.4 Experimental Results

In this section, we present some experimental results using the prototype. There are three classes of experiments presented. In Section 9.4.2, we attempt to replicate some of the simulation experiments presented in the earlier chapters on the prototype. The goal of these studies is to validate the prior assumptions and to investigate the accuracy of the simulation in modeling a broadcast environment. Then, we quantitatively explore some issues which were beyond the scope of the simulations.

On the client-side, we investigate two factors affecting performance. While the earlier simulation studies provided insight into the fundamental characteristics of the broadcast environment, the following two conditions were implicitly assumed:

1. *No cache management overhead.* In the simulation studies, the sole criteria to compare the various cache management algorithms was their response time. However, each of the caching and prefetching algorithms have different complexity and impose differing processor and network overhead at the client. For example, prefetching algorithms require the broadcast to be sampled continually which is not necessary for demand caching. Due to absence of data to accurately measure this overhead, the simulation studies treated all algorithms equally (for that purpose).[3]

2. *Clients never miss any broadcast page.* In the simulation, the client controlled the rate at which pages were sent out from the server. Consequently, the clients never "dropped" any page on the broadcast channel and always received pages in the order dictated by the broadcast program.

However, in a real implementation, the above assumptions will not hold and in Section 9.4.3, we compare the robustness of the various caching algorithms when that happens. First, we study the effect of the actual system load imposed by the various algorithms on their performance. Also, in a real system, if a client does not retrieve data from the channel fast enough to keep up with the broadcast, there exists a potential for it to miss pages. Thus, the experiments in Section 9.4.3 also compare the robustness of the various algorithms to missing data.

On the server-side, we investigate the utility of the cache at the server (Section 9.4.4). For the machines used in the experiments, the maximum data transfer rate from the disk was much lower than the maximum bandwidth of the ethernet network.[4]. Since the data has to move from the server disk, where it is stored, to the network, where it is disseminated, this experiment aimed at exploring the utility of the server cache in mitigating the impedance mismatch between the disk and the network throughput. Recall that the simulations made an implicit assumption that the entire database was cache-resident at the server.

9.4.1 Parameter Settings

As in the simulation experiments, the primary metric of comparison was the client response time. Since data was broadcast over the ethernet, there were two units of measurement —

[3]While the simulation studies did not account for the overhead, the problem was not ignored. $\mathcal{LIX}$ and $\mathcal{APT}$ were proposed as cheaper alternatives to the "pure" $\mathcal{PIX}$ and $\mathcal{PT}$ algorithms respectively.

[4]In the current technology, this is also true for most off-the-shelf disks and networks.

Name	Description	Setting
PageSize	Size of a page broadcast	1024 bytes + 76 bytes (header)
NetworkType	*SlowNet*	10 Mbit/sec (max) Ethernet
	FastNet	100 Mbit/sec (max) Ethernet
ServerCacheSize	Size of the server cache	50-3000 pages

Table 9.1: Prototype Parameter Description

Broadcast Units which was also used in the simulations and *seconds* which measured the performance in real time. Since one *Broadcast Unit* is the amount of time required to transmit a page, knowledge of the network bandwidth and the size of each page also allowed us to also independently validate one measurement from the other. Note that unlike the simulations, the response time (in seconds) truly reflects the cost of the algorithm including any cache management overhead. The working of the client and the server was exactly as in the simulator (Chapter 4).

The server was an Intel machine with a 200 Mhz Pentium Pro processor and 96 MB of memory. The client machines had 200 Mhz Pentium processors with 64 MB of memory. They were connected together via either 10 Mbit/sec or 100Mbit/sec ethernet network and were running Windows NT(4.0). The client runs a loop requesting pages using a skewed Zipf Distribution while the server broadcasts pages in the order dictated by the Broadcast Disks program. The server attempts to send out pages at the maximum rate possible. However, the actual rate of the data transfer is strongly dependent on the bandwidth of the ethernet connection and the overhead of the operating system. The graphs that follow present only a small subset of the results generated. Further details on the various experiments and the methodology used to generate them is available in [Cui97].

In the studies described next, unless otherwise stated, we choose the values of the parameters exactly as in the corresponding simulation experiment. The three parameters which do not have analogous equivalents are shown in Table 9.1. The *PageSize* parameter gives the size of the page in number of bytes and was set at 1000 bytes for all the experiments. Also, as shown in Figure 9.2, in addition to the data, each page also has a header. The size of the header was 76 bytes.[5] The second parameter is the *NetworkType*, which determines if the *SlowNet* (10Mbit/sec ethernet) or the 100 Mbit/sec *FastNet* was used for the experiment. The last parameter denotes the size of the cache at the server. For the experiments in Sections 9.4.2 and 9.4.3 it is set to 1000 pages. In Section 9.4.4 which studies the influence of the server cache, this parameter is varied from 50 to 3000 pages.

[5]Use of various optimizations can reduce the size of the header significantly. Since bandwidth was not as issue, reducing space overhead was not a priority.

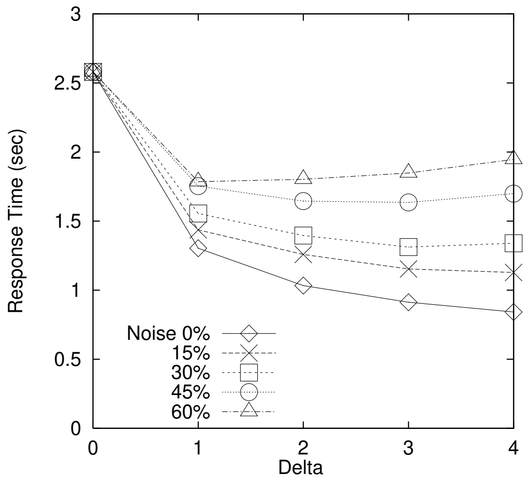


Figure 9.4: Noise Sensitivity
3-Disk (<300,1200,3500>), *CacheSize* = 1

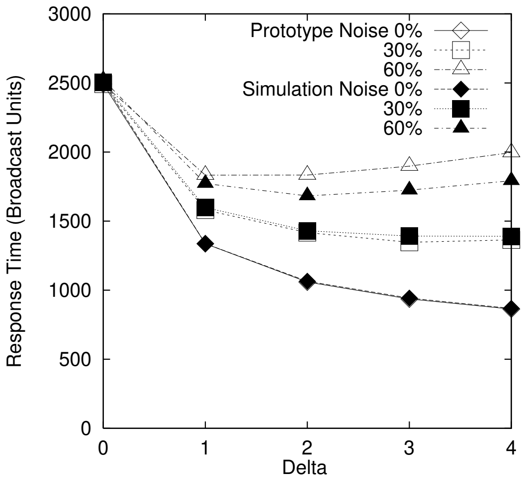


Figure 9.5: Response Time Comparison
3-Disk (<300,1200,3500>), *CacheSize* = 1

9.4.2 Simulation Validation

In this subsection, we present experiments which compare the performance of the simulator and the prototype under similar conditions. As pointed out above, the simulation studies are valid only when the clients do not drop any of the broadcast pages. Thus, for these set of experiments, the *SlowNet* channel (10 Mbit/sec ethernet) was used. Due to the various system overheads at the server and due to limitations of the operating system, only about 75% of the available bandwidth was taken up by the broadcast.[6] This bandwidth was low enough for the clients to handle the traffic without dropping any pages. Consequently, the simulation pre-condition of no page drops was replicated in the prototype allowing for a fair comparison between the results from the two setups. The goal of these experiments was to validate the various simulation assumptions and to also verify the accuracy of the numbers presented in the earlier chapters. We conducted the validation experiments in two phases — first, the robustness of the server was tested followed by experiments validating the client-side results.

Server Validation

To compare the server code on both the setups, we ran experiments in which the clients performed no caching. These experiments correspond to those presented in Section 5.2.2. In the absence of any caching effects, the performance numbers are a direct reflection of the server broadcast program.

Consider Figure 9.4 which presents the results of replicating the experiment in Figure 5.3

[6]The average bandwidth varied from 7.5-7.8 Mbit/sec. Section 9.4.4 explores why the server fails to utilize the entire bandwidth.

on the prototype. It studies the performance of a cacheless client for different *Noise* and Δ values for a 3-Disk broadcast and 5000 page database. The other parameters have exactly the same settings as used in the simulator (Table 5.1).

The shape of the curves in Figure 9.4 (in seconds) resemble those in the analogous simulation graph (Figure 5.3) which reports the numbers in *Broadcast Units*. Since there exists a linear correlation between the two units, it implies that the client behaviors are qualitatively similar in the two setups.

To see how closely the numbers compare quantitatively, Figure 9.5 plots the performance for three *Noise* values from the prototype and the simulations. Note that the response time in this graph is given in *Broadcast Units* which allows for a direct comparison. The graph shows that the two results follow each other closely for low *Noise* values ($<$ 1% variance for *Noise* of 0% and 30%) but deviate once the *Noise* becomes high (at 60% *Noise*, the average difference is about 6%). This is a consequence of the different ways by which *Noise* is generated in the prototype and the simulator.[7] In the simulator, since only a single client was run, the client access pattern was systematically perturbed as dictated by the *Noise* algorithm presented in Section 4.2.1. In the prototype, due to the need to support multiple clients, instead the server access pattern was permuted before setting up the broadcast (leaving the client access patterns intact). Generating *Noise* on the server-side causes slightly higher displacements than in the client-side, showing up in poorer performance for the prototype for higher *Noise* values.

Nonetheless, the graph confirms the accuracy of the simulation results and reinforces the lessons from the earlier studies — while a cacheless client with a skewed access to the database benefits from a multi-disk broadcast, it is sensitive to vagaries of the broadcast (as seen for the higher *Noise* values).

Client Validation

On the client-side, we performed similar tests on the prototype to explore the performance of various caching algorithms. As in the last set, the environment was tailored to mimic the simulation conditions. The performance of the four caching algorithms introduced earlier — $\mathcal{P}$, $\mathcal{PIX}$, $\mathcal{LIX}$ and $\mathcal{PT}$ were studied. Not surprisingly, the response time numbers were very similar to that in the simulator. This is to be expected since the client and the server were operated under similar conditions in both the setups.

Figure 9.6 shows the performance of $\mathcal{PT}$ and $\mathcal{PIX}$ for experimental settings used to generate Figure 6.10 (Table 6.2). For a 3-disk broadcast and a skewed client access, it shows the sensitivity of the two algorithms for changes in Δ. The experiment was run for *Noise* of 30% and for a client cache of 500 pages. As the graph highlights, there is a strong overlap between the lines for the simulator and the prototype; however the numbers for $\mathcal{PT}$ have slightly lower

[7] The details of the difference between the two *Noise* algorithms is beyond the scope of this chapter and is presented in [Cui97].

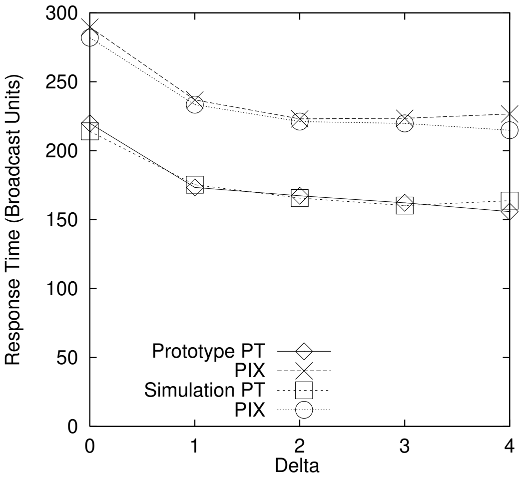


Figure 9.6: Caching Performance
3-Disk (<300,1200,3500>), *CacheSize* = 500

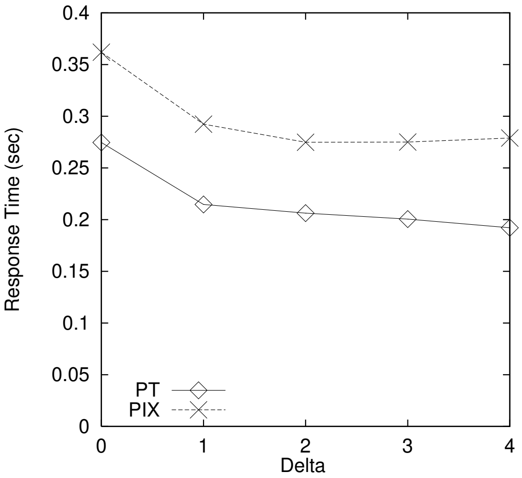


Figure 9.7: Client Response Time
3-Disk (<300,1200,3500>), *CacheSize* = 500

variance between the two setups than for $\mathcal{PIX}$ (the worst case difference is 5% of $\mathcal{PIX}$ and 3% for $\mathcal{PT}$). The lower variance for $\mathcal{PT}$ is due to its better robustness to non-ideal broadcast program than $\mathcal{PIX}$ (Figure 6.8); recall that the perturbation of the pages in the prototype is slightly higher than in the simulations due to different techniques to generate *Noise*. A similar overlap was observed for the other caching algorithms as well.

Figure 9.7 shows the response time for the two algorithms in seconds. For this set of experiments, we achieved a broadcast bandwidth of about 6.8 Mbit/sec and this number is validated by comparing the response time numbers in the two units. Since the broadcast bandwidth remains nearly constant for all values of Δ, the shape of the curves are similar to the lines on the left. The data is being broadcast on the slow network in this case. The real-time numbers can be expected to improve once the rate of transfer increases since more pages can be broadcast per unit (real) time. The performance of the two algorithms on the faster network will be studied in the next set of experiments.

The graphs in this section demonstrate that the simulation results accurately model the performance characteristics of a dissemination-based environment, subject to the constraints listed earlier being satisfied. In other words, as long as the broadcast rate is slow enough not to overload the client, the numbers for the simulation experiments can be used to faithfully characterize a broadcast environment (for the simulation space studied).

9.4.3 Cache Management Overhead Experiments

The studies presented in the previous chapters highlighted a number of unique characteristics of the broadcast environment. However, one of the shortcomings of the simulation model was its inability to capture the *actual* cost of client and server processing. For example, in Chapter 6, it was shown that the prefetching heuristic $\mathcal{PT}$ easily outperformed the demand-driven algorithm $\mathcal{PIX}$. However, the fact that $\mathcal{PT}$ has a significantly higher complexity than $\mathcal{PIX}$ was not reflected in the result because the model of the client did not account for the cost of client processing. $\mathcal{PT}$'s algorithmic complexity comes not only because each prefetching decision takes O(*CacheSize*) operations as opposed to O(log(*CacheSize*)) for every replacement decision in $\mathcal{PIX}$[8] but also because the heuristic has to be evaluated every time a *page is broadcast* as opposed to only *on a cache miss* for $\mathcal{PIX}$. In addition, prefetching requires that the broadcast be monitored continually and thus, imposes additional network-level overhead which was not modeled either.

The prototype allows us to study the working implementations of the various caching algorithms and thus, provides another dimension for comparison. In this section, we rate the $\mathcal{PIX}$ and $\mathcal{PT}$ strategies based on their true performance cost and study the tradeoffs between algorithmic overhead and performance. We also move out of the niche domain of the simulation studies and explore the robustness of the two algorithms when the broadcast rate is high enough to swamp the clients.

In Figures 9.6, we demonstrated that for low network bandwidths, the performance numbers of $\mathcal{PIX}$ and $\mathcal{PT}$ are comparable to those from the simulations. Thus, if pages are not broadcast at a very high rate, the processor (and the system in general) is fast enough to keep up with the broadcast. Thus, even though $\mathcal{PT}$ has a significantly higher listening and processor overhead than $\mathcal{PIX}$, it still provides superior performance due to the little penalty imposed by the listening. However, as the broadcast bandwidth improves, the slack available to make each caching (or, prefetching) choice shortens, eventually causing poorer performance due to swamping of the client.

To this end, we repeated the last experiment on the *FastNet* channel (100 MBit/sec ethernet). Due to server overhead and operating system limitations, the effective bandwidth used by the broadcast was only about 16.8 Mbit/sec which was 2.5 times more than the speed of transmission under the *SlowNet* used earlier.[9] In Figures 9.8 and 9.9 , the performance of $\mathcal{PIX}$ and $\mathcal{PT}$ are compared for the two network bandwidths. In Figure 9.8, the performance is given in *Broadcast Units* while the graph on the right shows the numbers in real time units of seconds.

Figure 9.8 shows that the performance (in *Broadcast Units*) of both $\mathcal{PIX}$ and $\mathcal{PT}$ are immune to the bandwidth of the broadcast channels under consideration.[10] Of course, due to the

[8]This assumes that the *pix* values for the cache-resident pages remain constant and thus, can be maintained as a sorted list.

[9]The next section explores why the entire available bandwidth is not utilized by the broadcast.

[10]Recall that this unit counts the *number* of pages sampled per miss and not the total time.

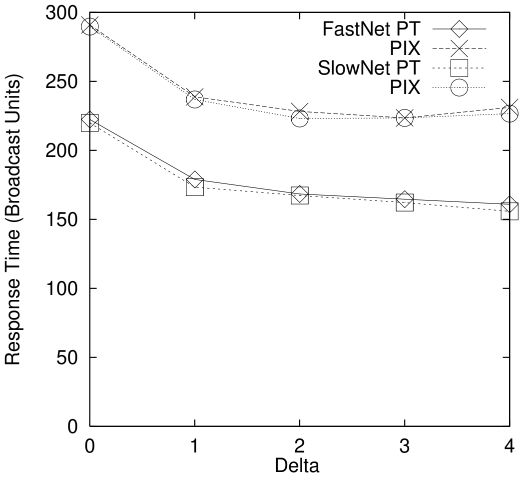


Figure 9.8: Performance vs. Bandwidth
Logical Time, *CacheSize* = 500, *Noise* = 30%

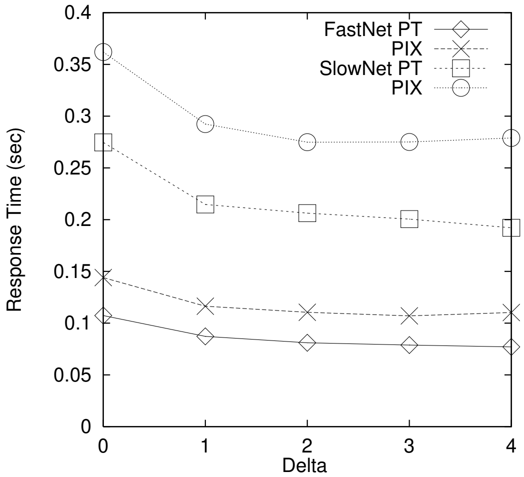


Figure 9.9: Performance vs. Bandwidth
Real Time, *CacheSize* = 500, *Noise* = 30%

higher bandwidth of the *FastNet*, both algorithms do significantly better (in real-time in seconds) than on the slower network by a factor equivalent to the bandwidth ratios (Figure 9.9).[11] Similarly, both $\mathcal{LIX}$ and $\mathcal{APT}$ which are cheaper approximations to $\mathcal{PIX}$ and $\mathcal{PT}$, were found to have exactly the same behavior. This is surprising and demonstrates that even though the faster network imposes stiffer constraints on the client, the broadcast does not overload the system enough to affect performance.

To verify this conjecture, we compared the percentage of the processor used by each of the four algorithms. This was done by taking the ratio of the total user and kernel time taken by the algorithm to the total time taken (at the client) to complete a single experimental run. This is shown in Figure 9.10 for the slower ethernet and in Figure 9.11 for the faster ethernet. In both the histograms, the processing overhead of solely "listening" to the broadcast is shown as a line parallel to the x-axis. This value was generated by making the client run without making any requests for pages and without any prefetching. Thus, every page is received by the *Network Manager* (Section 9.3.2) and immediately discarded. In effect, this is the cost of extracting the data from the multicast packet, moving up the layers of the network stack, interrupting the processor and any associated context switches. Note that this is the *minimum cost* every caching algorithm at the client has to bear.

The two sets of results validate a number of our intuitions listed earlier. As expected, the prefetching algorithms ($\mathcal{PT}$ and $\mathcal{APT}$) entail significantly higher processor overhead than the demand-caching $\mathcal{PIX}$ and $\mathcal{LIX}$. Each of the approximations $\mathcal{LIX}$ and $\mathcal{APT}$ also have a lower overhead than their "pure" counterparts, $\mathcal{PIX}$ and $\mathcal{PT}$ respectively. The overhead on

[11] For these experiments on the *FastNet*, negligibly small percentage of pages were dropped by the client network interface. The effect of page drops on performance will be studied in the next set of experiments.

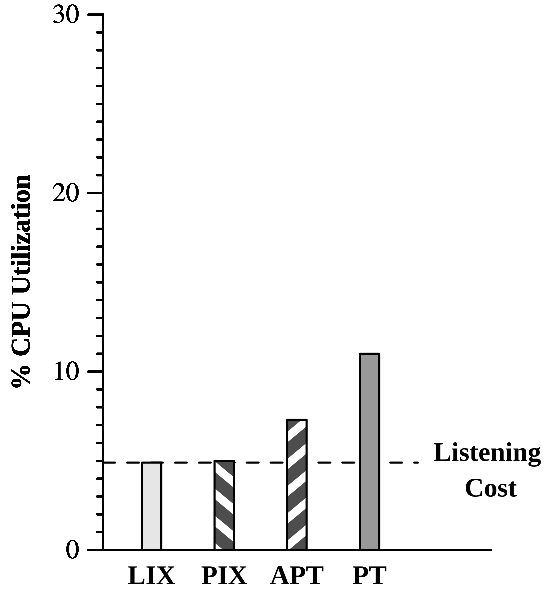


Figure 9.10: Caching Overhead, *SlowNet*
Noise = 30%, *CacheSize* = 500

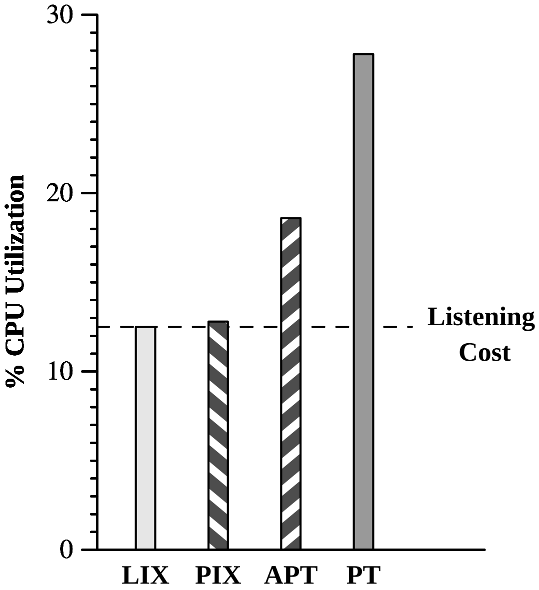


Figure 9.11: Caching Overhead, *FastNet*
Noise = 30%, *CacheSize* = 500

the slower ethernet is also much lower than that for the faster network because data appears much less frequently on it. In fact, the ratio of the cost of solely listening on both the networks is exactly the same as the ratio of their bandwidth (12.5% / 4.9% CPU $\approx$ 2.5).

Turning to the specific algorithms, nearly the entire cost of the demand-caching algorithms $\mathcal{PIX}$ and $\mathcal{LIX}$ are due to the operating system overhead of monitoring the broadcast. The listening overhead accounts for nearly 98% and 99.5% of the total cost of $\mathcal{PIX}$ and $\mathcal{LIX}$ respectively. Thus, if the execution of the network-level calls provided by the operating system can be further optimized, these algorithms can expect a dramatic improvement in performance.[12] This also points to the fact that the motivation behind the design of $\mathcal{LIX}$ was misplaced. While $\mathcal{LIX}$ was shown in Chapter 5, to be a good match for $\mathcal{PIX}$, it was designed as a cheaper (in terms of complexity) alternative to $\mathcal{PIX}$ at the expense of trading performance. Obviously, $\mathcal{PIX}$ is an ideal algorithm which requires the knowledge of probability of access and cannot be implemented in practice. Since the algorithmic overhead of $\mathcal{PIX}$ is very low, the lesson from these histograms is that it is more worthwhile to focus on designing better approximations to probability as opposed to reducing algorithmic complexity. Developing such algorithms is an interesting avenue for future study.

However, the same lesson is not valid for the prefetching heuristics $\mathcal{PT}$ and its approximation $\mathcal{APT}$. $\mathcal{APT}$ reduces the utilization by more than one third making it an attractive alternative. While load on the processor is not an issue for the setup studied, as pointed out

[12]This observation also hold for the prefetching algorithms though the improvement in that case would be less spectacular.

above, once greater bandwidth can be extracted out of the network, $\mathcal{PT}$'s significantly higher overhead would make it ineffective.

The general observation from the two graphs is that the effect of the algorithms on the system load is minimal and thus, unlikely to impact the performance of other processes on the client. This also explains why the performance numbers for $\mathcal{PIX}$ and $\mathcal{PT}$ were the same (in logical units) for the slow and the fast network (Figure 9.8).

The results in this section point to the fact that for the experimental space studied, the hardware and processing capabilities of the machines used in the prototype easily overcome the resource requirements of the cache management algorithms. However, the scalability of the client to larger real-life database sizes needs to be investigated further. Since we were interested in understanding the tradeoffs, the experimental space in the studies was controlled in various ways. Firstly, when generating the results, the processor was dedicated to the caching algorithm by not running any other applications on the client. The next subsection briefly discusses the effect of the interference from other applications on the caching algorithm and those results show that overloading the processor can cause degradation in performance. Also, the studies presented were for a cache size of 500 pages. If the cache size was increased by an orders of magnitude (along with a similar increase in the database size), the overhead of updating the caching structures can be expected to be significantly higher. The client also benefited from the various system bottlenecks on the server-side. In spite of 100 Mbit bandwidth available on the *FastNet*, the server managed to utilize less than a fifth of its capacity. Since the numbers show that the processor utilization increases linearly with the broadcast rate for all the algorithms, utilization of the entire 100 Mbit bandwidth on the fast network, would rapidly saturate the processor. Thus, these results while highlighting important characteristics of the broadcast environment, require further validation on larger scale environments. However, under the conditions studied, the differences in overhead failed to have any impact on the performance in real-time.

Robustness to Page Drop

The simulation subsystem studied the Broadcast Disks environment in an ideal setting. The effect on client performance if it failed to sample every broadcast page was not investigated there. In a real system, this can happen for at least two reasons: a) if the rate of broadcast is higher than the rate at which the client can retrieve data from the network, pages can be lost due to overflow of the client's network interface buffers and b) it the channel is noisy, data may be corrupted in transmission. In this subsection, we do a sensitivity analysis on the performance of the demand caching algorithm $\mathcal{PIX}$ and the prefetching algorithm $\mathcal{PT}$ to page drops.

In order to be able to conduct this experiment, we had to use an external load to saturate the processor on the client and in effect, slow down the rate of retrieval from the client's network interface. This is because even on the fast ethernet, the rate of data transmission ($\approx$ 17 Mbit/sec) was within the tolerance of the client network and processor resources. Thus, an extremely small percentage of pages were dropped on the client-side for both the slow and the fast ethernet. Consequently, we were required to run a synthetic workload to attempt to overload the processor.

We first attempted to saturate the processor by running infinite loops on the client machine during the experimental run. Though two such loops immediately raised the cpu utilization to 100%, the caching performance was unaffected even when 50 loops were active simultaneously. We speculate that this is due to the bursty requirement of the processor by the caching algorithm (triggered by the arrival of a new page on the ethernet). Since the infinite loops are processor-intensive, their scheduling priority reduces with time[13] causing them to give up the processor when the caching algorithm is activated by the arrival of a new page. Additionally, the Windows NT scheduler may also favor network-based and disk-based processes much like most UNIX schedulers which favor interactive processes. Due to the unavailability of documentation on the internal details of the scheduler, we do not have additional evidence to corroborate these observations.

Based on the above conjecture, we then ran a synthetic workload of increasing number of video clips[14] to overload the system. The bursty I/O required to play a video interfered with the bursty requirement of the processor by the cache management routines causing pages to be dropped. The server augmented the broadcast by adding a counter to every page transmitted and receiving pages out of sequence indicated to the client that it dropped pages. The percentage drop was calculated as the ratio of the number of dropped pages to the total number of pages received by the client in a single experimental run.

Figures 9.12 and 9.13 show the percentage of pages dropped and the variation in client response time respectively as the load on the system is added by increasing the number of video clips displayed.

As the load is increased, Figure 9.12 shows that both algorithms start to drop pages. On the left end of the graph, we are in the ideal conditions as in the earlier prototype experiments and the simulation studies (i.e., negligible page drop, unloaded system). However, as the system load increases moving to the right, the bursty requirements of the processor by both the caching algorithm and the client's graphics interface displaying the clips start to conflict. This reduces the rate at which data is extracted from the channel by the caching algorithm causing an overflow of the network-level buffers leading to greater page drops.

There are two important observations to be made. One, the drop count for $\mathcal{PIX}$ remains

[13] Typically, schedulers decrease a process's priority in some proportion to the cpu time it has already consumed.

[14] A 23 second clip in QuickTime format which played in a continuous loop with a display size of 1.5x2.5 square inches.

unchanged until three or more clips are played whereas $\mathcal{PT}$ immediately starts to drop pages. This is due to the extra complexity of the $\mathcal{PT}$ algorithm which requires more processor time as shown in Figure 9.11. Since the computational requirements of $\mathcal{PIX}$ (total utilization minus listening cost) is negligible compared to $\mathcal{PT}$ (0.3% of the processor for $\mathcal{PIX}$ to 15.3% for $\mathcal{PT}$), the processor contention occurs at a much higher load than for $\mathcal{PT}$. The second point is that once both the algorithms starts to drop pages, the rate of drop increases linearly at roughly the same rate for both of them. This is because beyond a certain point, the processor is so saturated that the minor overhead differences between the two algorithms are no longer relevant.

Figure 9.13 shows the performance degradation of both the algorithms as the load on the processor is increased. On the left end of the graph, $\mathcal{PT}$ outperforms $\mathcal{PIX}$ since this is the ideal no-load setting. However, moving to the right, as more pages are dropped by the client, $\mathcal{PT}$'s performance degrades much more rapidly and eventually becomes worse than $\mathcal{PIX}$ midway on the x-axis. Quantitatively, comparing the impact of the loss of data on the performance shows that $\mathcal{PT}$ looses 16% of its performance for every percent drop in pages whereas $\mathcal{PIX}$'s response time increases by just 1.7% for the same drop in pages. The percentage drop at three clips when the order inverts is about 3.7% for $\mathcal{PT}$ and 1.75% for $\mathcal{PIX}$.

This demonstrates the frailty of any prefetch-based scheme. A time-based scheme like $\mathcal{PT}$ uses the *instantaneous* latency of a page to cycle data in and out of the cache. This was seen in the saw-tooth behavior of the `pt` value in Figure 6.1. Thus, $\mathcal{PT}$ creates cache real-estate by dropping pages just before they appear on the broadcast (when their instantaneous cost of access is low). However, while in the ideal scenario, it is guaranteed to fetch them back immediately afterwards, this no longer holds true once pages start to get dropped. $\mathcal{PIX}$, on the other hand, uses the *average* cost of access to make replacement decisions and therefore, important pages eventually settle into the cache permanently. Thus, in the best case (on the left end of the graphs), the fine-tuning by $\mathcal{PT}$ gives it nearly a 28% improvement in the response time over $\mathcal{PIX}$. However, once the client starts to drop pages, data no longer appears on the channel as expected (from the client's viewpoint), quickly degrading $\mathcal{PT}$'s performance (it is 55% worse at the extreme right). The performance particularly suffers when very important pages are dropped from the cache on the expectation that they would appear soon and be fetched back. This reinforces the notion that aggressive cache management like that done by time-based prefetch can be a double edged sword. Thus, in a real system, its use should be tempered based on the computing and network resources of the client.

Figure 9.13 also points to the need to develop dynamic algorithms which switch from aggressive prefetching to conservative demand caching (and vice versa) as need be depending on the prevailing conditions. In other words, the client would prefetch as long as the drop percentage was below a certain threshold and then automatically switch to a demand-caching strategy as the drop count increases. Such a scheme would be particularly useful in mobile wireless environments which are prone to very bursty error which implicitly create the effect

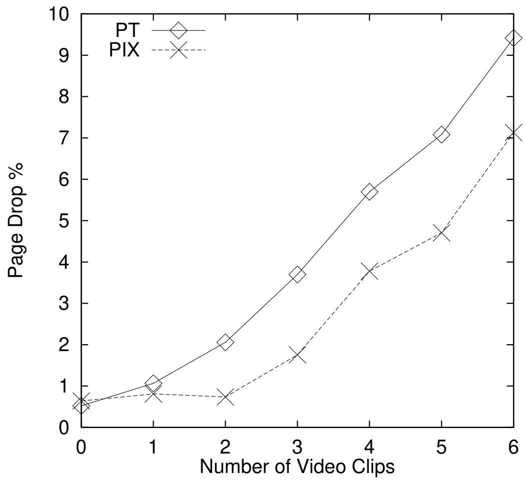


Figure 9.12: Page Drop% vs. System Load

Δ=2, *Noise*=30%

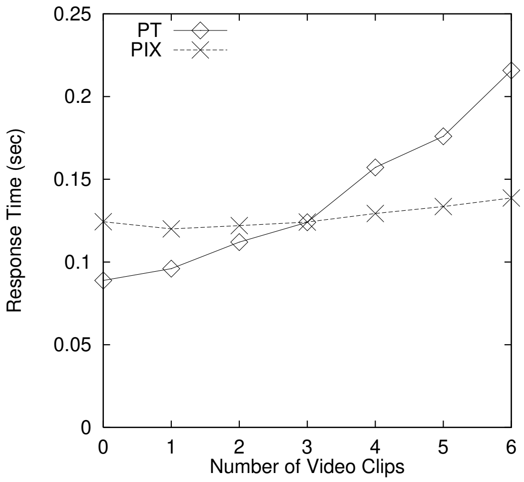


Figure 9.13: Effect of Page Drop

Δ=2, *Noise*=30%

of dropping pages.

9.4.4 Server Cache Experiments

In this section, we investigate the role of the server cache in a broadcast environment. In recent years, rapid improvement in network technology has caused the network bandwidth to easily surpass the data retrieval rate from disks. For dissemination-based systems, where the server has to continually move data from the disk to the network, this creates an impedance mismatch. Thus, the server cache is critical to the data transmission rate since it acts as an intermediate faster buffer between the slow disk and the fast network.

For this study, we used the same database size (3000 pages) and the broadcast program (3-Disk, <300,1200,1500>) as in the last set of experiments. The server cache was varied from 50 pages to 3000 pages (i.e., 1.6% to 100% of the database) and the experiments were run on the *FastNet*. To investigate the effect of caching a page, we counted the number of times a page was accessed from the disk (*Disk_In*) and the number of times it was broadcast (*Network_Out*) for the length of the experimental run. The ratio of *Network_Out* to *Disk_In* is referred to as the *NODI* ratio.

Before we describe the experiments, it is worthwhile to briefly explain the server caching strategy called *Farthest Time to (next) Broadcast* (FTB) used for these experiments.[15] FTB evicts the cache-resident page whose wait until the next transmission is the highest. If a flat broadcast is used (i.e., Δ=0), this implies that pages enter and leave the cache in a FIFO fashion.

[15] We only explain the algorithm for pages on the regular push schedule. The details of how pull pages requested via the backchannel and updated pages are handled is available [Bos97].

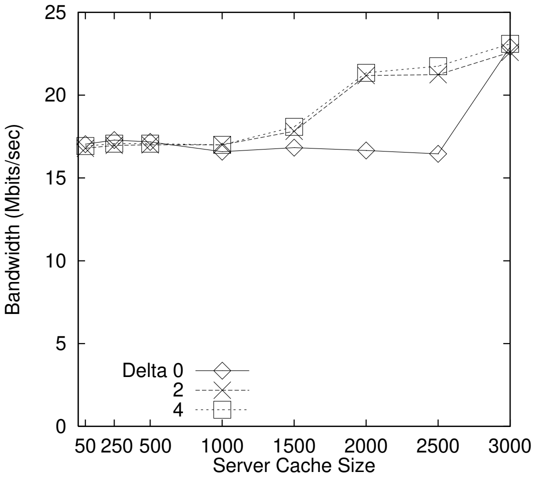


Figure 9.14: Bandwidth vs. Cache Size

Slow Ethernet, *CacheSize* = 500

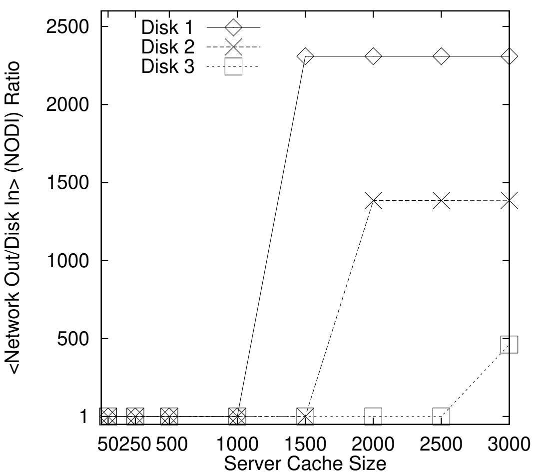


Figure 9.15: Server Cache Residency

Fast Ethernet, *CacheSize* = 500

However, for a multi-disk program, the cache is put to better use since different pages have different broadcast periods.

Figure 9.14 shows the bandwidth delivered by the server for increasing cache sizes for a flat broadcast (Δ = 0) and two skewed broadcast programs (Δ = 2,4). The graph shows that the bandwidth utilized by the broadcast remains more or less constant for all the three Δ values until a cache size of 1500 pages when the non-flat broadcasts improve their bandwidths by about 5% and 6.5% for Δ of 2 and 4 respectively. Then, there are two abrupt jumps in bandwidth at cache size of 2000 pages and finally, when the entire database is cache resident at which point the bandwidth peaks to about 23 Mbit/sec for all the Δ values.

The cause of these sudden improvements in the transmission bandwidth can be seen in Figure 9.15. It shows the *NODI* ratio for a sample page from each of the three disks. Recall that this ratio is a reflection of how long a page is cache-resident every time it is brought in from the disk and thus, a reflection of the disk access overhead savings. This graph is for the intermediate skewed broadcast (Δ = 2) with the server cache size on the x-axis. Recall that for this value of Δ, pages on the first disk are broadcast five times a cycle and pages on the second and the third disk are broadcast thrice and once respectively.[16] Thus, the size of a complete cycle of the broadcast (major cycle) is 6600 pages (using the algorithm in Section 3.2.2). Therefore, pages on the first and the second disk go out once every 1320 (6600/5) and 2200 (6600/3) pages respectively. Consider a page n on the first disk. If the server cache size is less that 1320 pages, FTB will evict n from the cache after broadcasting it only once because there are more pages within two transmissions of n than there are cache slots. However, once the cache size

[16] This 5:3:1 ratio is also reflected on the right end of the x-axis where the *NODI* values are at a maximum for all the three disks.

increases beyond that number, FTB fixes pages from the first disk permanently in the server cache. Thus, the *NODI* values for the first disk jumps from 1 to about 2300 (i.e., permanently cache-resident) at a cache size of 1500 pages and from that point on, no disk accesses are required for those pages. The reduction in the data retrieval overhead from the disk for these pages causes the first spike in the delivery bandwidth at 1500 pages (Figure 9.14).[17] The next jump occurs when the entire second disk becomes cache resident between a cache size of 1500 and 2000 pages.[18] Pages on the third disk, which go out only once a cycle become permanently cache resident only when the size of the cache equals the size of the database (extreme right end of the graphs) and this causes the final jump in the delivery bandwidth.

A surprising observation is that when the entire database is cache resident (at cache size of 3000 pages), the server barely manages to utilize a quarter of the available bandwidth of the network.[19] This shows that the disk is not the only source of the bottleneck but that the entire data and control path through the various system layers from the disk to the network interface are not efficient enough to keep up with the network speed and thus, also contribute to the poor rate of data delivery. Investigating the potential sources of system saturation (for the operating system/ hardware platform studied) and identifying means to eliminate them is an interesting area of further research and is beyond the scope of this current study.

9.5 Conclusions

In this chapter, we described the design and implementation of the Broadcast Disks prototype and explored the characteristics of a dissemination-based environment quantitatively using the prototype.

The motivation for building the prototype was to not only investigate the feasibility of building a dissemination-based system using off-the-shelf hardware components and software components but to also validate the various assumptions made in the simulation modeling. For the experiments in which the simulation conditions were replicated on the prototype, the numbers on the two setups were found to be within a few percentage points of each other. This confirmed that the simulation results presented in the earlier chapters can be faithfully used to describe the characteristics of a broadcast environment.[20]

[17] A similar argument works for Δ of 4.

[18] This actually occurs beyond a cache size of 1900 pages. Since each page from the first disk goes out at least twice for every page on the second disk, a cache size of 2200 (repetition period of a second disk page) minus 300 (size of first disk) = 1900 pages suffices to freeze the second disk in the cache.

[19] These numbers were also borne out by use of *TcpSpeed* [Max97], an utility which determines the maximum bandwidth of a TCP connection between any two networked machines.

[20] Of course, this is subject to the assumptions set forth in the simulations listed earlier being valid. These include no page drops at the client and a low broadcast rate which does not overload the client processing resources.

However, the prototype experiments also widened the scope of exploration by addressing a number of new issues not included earlier. On the client-side, they focussed on investigating the real-cost of the various cache management strategies including any algorithmic overhead. On the server-side, the impact of the server cache on the rate of data transmission was studied.

The client-side experiments found that the cost of continually monitoring the network, which is the minimum cost that needs to be paid, accounted for a big chunk of the total cost of all algorithms. As our previous intuition held, the load imposed by the prefetching algorithms were significantly higher than that for the demand-driven algorithms. For example, the processor utilization of $\mathcal{PIX}$ was less than half of the prefetching heuristic $\mathcal{PT}$. However, in spite of the differences in the overhead, none of the algorithms managed to saturated the processor and the network heavily enough to adversely affect performance. Consequently, the prototype results matched those from the simulation studies and in spite of its excessive overhead, prefetching was found a worthwhile approach. Another client-side study investigated the robustness of the various algorithms when the client failed to keep up with the broadcast rate and consequently, dropped pages from the network. It was found that even though $\mathcal{PIX}$ and $\mathcal{PT}$ missed pages at nearly the same rate under a synthetic external load, $\mathcal{PT}$'s performance degraded at a significantly higher rate than $\mathcal{PIX}$. This was due to time-based caching performed by $\mathcal{PT}$ which makes incorrect decisions once the guarantee of the order of transmission of pages as dictated by the broadcast program is lost. This result points to the need for dynamic caching strategies which switch between aggressive prefetching and conservative demand caching based on the processor and network capabilities of the system.

Finally, the utility of the server cache in mitigating the impedance mismatch between the disk and network was investigated. It was found that the rate at which the server broadcast data improved with increasing server cache size due to fewer accesses from the disk. However, a surprising observation was that even when the entire database was cache resident, the server managed to utilize only a quarter of the available 100 Mbit/sec ethernet bandwidth. This reflects the fact that for the commodity hardware used in the experiments, both the disk throughput and the data and control path through the system layers between the disk and the network are bottlenecks and thus, are unable to keep up with the improving network bandwidths available today.

In summary, the simulation results were validated by the prototype and found to faithfully reflect the characteristics of a dissemination-based environment. The experience of using the prototype to stress various components of the system, however, gave us conflicting results. On the client-side, the processor and network resources were found to dwarf the requirements of the cache management algorithms (including the more expensive prefetching strategies). On the other hand, on the server-side, the processor and the various systems layers were saturated much earlier than the network. It was observed that even in the best case, the server machine barely managed to extract a quarter of the available bandwidth for the broadcast program.

Chapter 10

Conclusions

In this thesis, we have presented a dissemination-based approach to designing client-server systems. In this chapter, we summarize all the results presented. We then conclude with a discussion of the various directions to extend this work and the opportunities for future research in the area of push-based systems.

10.1 Summary of Results

This thesis studies the dissemination-based model for data delivery as a means to improve performance and scalability for many new emerging applications in client-server environments. Data dissemination is a push-based approach which uses a broadcast channel to deliver data to the client population. The motivation behind this fundamentally different approach was the realization that the traditional pull-based model of designing distributed systems suffers from an inherent scalability bottleneck leading to severe performance problems. The drawbacks of the pull-based model have been particularly acerbated by the dramatic changes occurring in the computing and communication landscape over the past few years.

In recent past, there have been two parallel trends which have changed the nature of distributed computing. One, the availability of high-bandwidth links via satellite and cable networks into the home (and on the road) and the tremendous growth of the Internet and the World Wide Web, has created an explosion in the scale of the global internetwork. Also, either by design or due to resource constraints, many of the emerging environments have the property that they are *asymmetric*, i.e., the bandwidth available from the server to the clients is significantly higher than the bandwidth in the reverse direction.[1] Two, spurred by the falling costs of data transfer, both monetarily and in terms of bandwidth, there is an emergence of a number of new and bold information-centered applications. These applications, such as

[1]The definition of asymmetry is broader than just *bandwidth* asymmetry. Section 2.1.1 gives the various flavors.

stock tickers and traffic information systems, often have to support extremely large number of clients. The fundamental problem that this thesis attempts to address is, given the emerging scale of the internetwork and the rise of asymmetric environments, what is the most efficient way to deliver data from servers to the clients?

In the early chapters of this thesis, we make a case for using a push-based approach to data delivery and motivate why these new trends inherently limit the utility of the traditional pull-based approach. Unlike, the request/response paradigm which forms the basis of the pull-based model, the data is delivered from the server without any explicit request from the client in a push-based system. In this thesis, we concentrate on one model for doing push, the *periodic dissemination* model. In this model, the server uses a broadcast channel to deliver data to all the clients simultaneously ("data dissemination") and it uses a cyclic schedule which repeats every broadcast *period*.

The *Broadcast Disks* model was proposed as a framework for periodic dissemination. The previous approaches in dissemination used a *flat* model wherein all data items were broadcast once a cycle. The Broadcast Disks approach, models the broadcast as a number of interleaved "disks", each of different sizes and spinning at different speeds. Using this formulation, the Broadcast Disks model is able to deliver data items to the clients at different rates commensurate with their utility to the client population. The bulk of this thesis concentrates on studying the utility of this multi-disk approach and investigates the tradeoffs using a simulation-based performance model and a working prototype.

The initial set of experiments studying the basic tradeoffs justified two of our fundamental intuitions: a) the multi-level approach of the Broadcast Disks framework is better suited to the skewed access in typical database applications and b) that the role of caching in this environment is fundamentally different from traditional pull-based systems. Since the data is broadcast over a serial channel, the cost of access of each item is not fixed. Also, the multi-disk nature of our framework delivers data to the clients with varying latency. Consequently, traditional probability-based caching algorithms, such as LRU, which implicitly assume an uniform cost of retrieval for all pages, were shown to perform poorly in this environment. We, instead, proposed a *cost-based caching* approach to cache management which took into account the expected cost of retrieval in the replacement decision. We presented two cost-based algorithms for demand-driven caching — an optimal algorithm $\mathcal{PIX}$ and an implementable approximation $\mathcal{LIX}$. As expected, they outperformed their traditional counterparts for all the cases studied.

We then extended the cost-based caching idea to prefetching. A dissemination-based environment is ideally suited to prefetching since data flows past the client creating greater opportunities for inserting and deleting pages from the client cache. Using a simple idea called Tag-team caching, we demonstrated that prefetching alone truly exploits the full potential of the broadcast environment. The key intuition here is that the replacement decision taking into account the *instantaneous* cost of access outperforms an algorithm that uses the *expected* cost (as was done in the demand-driven caching case). However, even though prefetching provides better performance, it entails a much higher overhead since maintaining the instantaneous cost accurately is significantly more complex that maintaining the expected cost. We proposed two prefetching heuristics — a pure algorithm called $\mathcal{PT}$ (which has been shown to be optimal under certain circumstances) and an implementable approximation $\mathcal{APT}$.

In this thesis, we also studied the influence of updates on the client performance. This study produced some surprising results when compared to how traditional pull-based systems react to the presence of updates. Two of the key contributions of the study were as follows: a) Unlike in traditional pull-based systems, the broadcast environment was found to be extremely robust in the presence of updates and very simple enhancements provided close to read-only performance and b) Propagation-based algorithms were found to deliver better client performance than invalidation-based algorithms. This is contrary to traditional systems where the inverse almost always holds true. We also studied the system performance for varying degrees of consistency. As in traditional systems, it was found that the stricter the consistency requirement, the poorer the performance.

The last set of simulation-based studies investigated the tradeoffs between using pull-based and push-based delivery. The study justified our intuition that the pull-based approach gave better performance for a lightly loaded system with few clients. However, with increasing size of the client population, the pull-based approach hits a scalability bottleneck and saturates the server, degrading the performance below a push-based approach. We then attempted an hybrid strategy to use both push and pull simultaneously in order to get the best of both approaches. The key issue was balancing the downstream bandwidth from the server to the client between the push and pull delivery. We concluded that efficiently dividing the bandwidth across the two delivery options, was critical in determining the client performance. We also investigated a few techniques to delay the point of server saturation in order to accommodate more clients in the system.

Finally, we designed and implemented a working Broadcast Disks prototype. Besides gaining valuable experience in building such a novel system, we hoped to study a number of factors that the simulation failed to capture. The validation studies, which replicated the earlier simulation experiments on the prototype, generated results which were found to be within a few percentage points of the simulation numbers. Thus, for the assumptions under which the simulation studies were valid, the simulation results could be used to faithfully characterize a broadcast environment. We also investigated the actual running cost of the various caching algorithms. They verified our earlier intuitions that prefetching heuristics, while providing better performance, also impose greater load on the system than demand-driven strategies. In fact, if the client fails to sample every page on the broadcast channel, the prefetching heuristic $\mathcal{PT}$ was found to quickly degrade in performance below $\mathcal{PIX}$ which demonstrated the frailty of aggressive prefetch-based schemes. Finally, we also explored the influence of the server cache of the speed at which the server could deliver data on the broadcast channel. While the cache improved the data delivery rate by reducing the disk access overhead, surprisingly the server failed to utilize the entire channel bandwidth even when the all of the database was cache-resident. This demonstrated that for the current technology, network bandwidths are significantly higher than the capabilities of the rest of the computing system (including the disks) to deliver data at the maximum rates. Thus, while clients can still be a source of bottleneck in a dissemination-based system by failing to keep up with a high-speed broadcast, we failed to demonstrate that as an issue for the setups studied.

10.2 Future Work

The area of data dissemination and push-based systems in general is still in its infancy and this work just touches the tip of the iceberg. The continuing growth of the Internet is spawning new push-based commercial applications in greater numbers. Also, frequent outages on the WWW during popular events like elections and the Olympics, only reflect the inability of a pull-based approach in supporting large-scale information centered applications.

To make the study of the Broadcast Disks model tractable, we restricted the environment in various ways and relaxing these constraints creates a number of new opportunities for further research.

One of the important areas of future study is the collection and processing of user profiles. User profiles are critical to utility of a push-based system since a push program is ineffective if it fails to reflect the user needs. Techniques to gather profiles automatically and algorithms to combine multiple profiles to extract useful information is a fertile avenue for research.

Another server-side issue which deserves greater attention is the generation of broadcast programs. In this thesis, we only provide rudimentary guidelines about the desirable properties of such programs. This area is already beginning to receive some attention as pointed to in Section 3.6. However, given the complexity of generating optimal programs, deriving efficient approximations which provide good client performance is an interesting challenge.

One of the implicit assumptions made in this work is that the communication channel is dedicated to the broadcast program and error-free. However, in real-life this is unlikely to be true. Another, fertile area of research is considering dissemination in error-prone environments such as wireless networks. A related problem is investigating the feasibility of dissemination in heterogeneous environments where different clients connect using channels of differing bandwidth.

From a database viewpoint, an immediate extension of this work is to augment the client to make it transactional. If dissemination-based database systems hope to expect to be as widespread as traditional distributed database systems, they have to provide the same level of database support as traditional systems. Thus, an interesting challenge is to study query processing and transaction support in the framework of a dissemination-based system.

On a broader scale, this work on Broadcast Disks sets the foundation for a novel approach to distributed information systems called *Dissemination-based Information Systems* (DBIS). DBIS was introduced in [FZ96] as a framework for distributed computing systems which allows multiple data delivery options including push and pull, periodic and aperiodic delivery, and so on. In the ideal scenario, a DBIS would provide a high-level toolkit for the user to specify the requirements. The DBIS would then automatically choose the best data delivery option and if need be, dynamically change the mode of data delivery to reflect changing access patterns or system load. Also, unlike in this thesis which considered only a single-level of client-server interaction, a DBIS allows for arbitrary levels with some nodes playing a dual role of being a client of and a server to different set of nodes. Designing and implementing such a system is an enormous challenge and requires not only new database solutions but also, contributions from many diverse fields. This thesis on Broadcast Disks provides a foundation to build from and highlights the various issues that need to be addressed in the design and implementation of dissemination-based systems.

Appendix A

Proof of Optimality of $\mathcal{PIX}$

In Section 5.4.1, $\mathcal{PIX}$ was introduced as a cost-based caching algorithm. In response to a *cache miss*, $\mathcal{PIX}$ evicts the page in the cache which has the lowest ratio of the access probability (P) to the server's broadcast frequency (X), i.e., P/X. In this section, we prove that under a demand-driven model of access, this heuristic used by $\mathcal{PIX}$ is optimal in minimizing the latency (defined next) of a cache miss in the Broadcast Disks framework. This argument will then be extended to prove that $\mathcal{PIX}$ provides the lowest overall data access cost and thus, the best response time for the client.

A program adhering to the Broadcast Disks framework is *regular* and *periodic* (Section 3.2). A program is said to be *regular* if consecutive occurrences of the same page have the same inter-arrival time and it is *periodic* if it has a fixed length after which it repeats. The proof outlined next is subject to the following conditions:

- *Clients do not prefetch.* The result is valid only for *on-demand* access from the broadcast. In other words, the broadcast is sampled only when the application accesses a page not in the client cache (i.e., a cache miss). In response to the miss, the client waits for the page to be broadcast and picks it up. It then ignores the broadcast program until the next miss.
- *Fixed-sized data items.* The server broadcasts the data in fixed-sized units called "pages".
- *Program is static.* A program is static if the order in which pages are broadcast remains unchanged across periods. While this is not a restriction imposed by the Broadcast Disk model, we have assumed this structure for the read-only studies in this thesis. A consequence of this restriction is that the length of every cycle, the *period*, is the same for all cycles.
- *Blocking access model.* We assume that in case of a cache miss, the the client processing is halted until the page accessed arrives on the broadcast. The time difference from when

the access is made until when the broadcast delivers the page is the *latency* or the *delay* of the access. We also assume that a cache hit has a zero latency. Thus, the cost of data access for any caching strategy is determined by the miss latency.

- *No prescient knowledge.* The caching algorithm has no knowledge of the future access pattern and makes its replacement decision solely on the knowledge available at that moment.

Let, D be the size of the database being broadcast, B the size of the cache at the client and T the the period of the broadcast program. For each page $i \in D$, let,

- p_i : the local client probability of access for page i.
- x_i : the number of times i appears per cycle, i.e., the server's broadcast frequency for i.
- l_i : latency of a cache miss for i; this is the time from when the access is made until the page arrives on the broadcast.

Since there is no prefetching, the cache contents change only if there is a cache miss. Let the client accesses be numbered sequentially. Let,

- a_k : the page requested on the k^{th} access
- C_k : the set of pages in the cache after the k^{th} access, $C_k \subseteq D$.

Thus, the cache state evolves as follows:

$$C_{k+1} = \begin{cases} C_k & : \; a_{k+1} \in C_k, & \# \; \text{(cache hit)} & (a) \\ C_k + a_{k+1} & : \; a_{k+1} \notin C_k, |C_k| < B, & \# \; \text{(warm up)} & (b) \\ C_k + a_{k+1} - v & : \; a_{k+1} \notin C_k, |C_k| = B, \text{Victim } v \in C_k. & \# \; \text{(cache miss)} & (c) \end{cases}$$

This is explained as follows. When a client joins the system, it starts with an empty cache. In the warm up phase when the cache is filling up, every cache miss results in a new page entering the cache (Case b). However, once the cache is full, a cache miss results in a victim being evicted to make space for the new page accessed (Case c). Of course, if the page accessed is already cache-resident, the cache state remains unchanged (Case a).

We are interested in the Case (c). The key decision for a caching strategy is the choice of v, the victim. To find space for $(k+1)^{th}$ accessed page a_{k+1}, $\mathcal{PIX}$ chooses as the victim the page $v,\ v \in C_k$ such that,

$$\forall j \in C_k,\ \frac{p_v}{x_v} \leq \frac{p_j}{x_j}.$$

We outline below how this choice of v, gives the best client performance.

The data access cost for any page is the product of the probability that it is accessed and its latency of access. As pointed out earlier, if the page accessed is cache resident, the latency is zero; else, it is the delay until the page is broadcast. Assume, without loss of generality that k accesses have taken place. Thus, the cost S_k of data access for any caching strategy after the k^{th} access is given by:

$$S_k = \sum_{i \in D} p_i.l_i, \quad \text{where } l_i : \text{latency of page } i. \tag{A.1}$$

The lower this cost, the better the caching strategy. Equation A.1 can be re-written as,

$$\begin{aligned} S_k &= \sum_{i \in C_k} p_i.l_i + \sum_{i \notin C_k} p_i.l_i, \\ &= \sum_{i \in C_k} p_i.0 + \sum_{i \notin C_k} p_i.l_i, \quad \text{(since a cache hit has zero latency)} \\ &= \sum_{i \notin C_k} p_i.l_i. \end{aligned} \tag{A.2}$$

We will derive the optimality of $\mathcal{PIX}$, by proving an invariant that the cost S_k using $\mathcal{PIX}$ is always the minimum for all k. The proof using induction is as follows:

- *Base Case*: $k = t$, S_t *is minimum cost.* t *is the lowest access for which* $|C_t| = B$.
  This case refers to the state of the cache when it is full for the first time after warm-up. The cost S_t is obviously optimal since no victim has been chosen as yet (i.e., only Case (a) and (b) of the cache transition have been used). Since the difference between caching strategies is how they choose a victim, all strategies (including any optimal strategy) would be at the same state C_t after t accesses.

- *Hypothesis Step*: *Assume* S_w *is the optimal cost for some* $k = w$, $w > t$.

- *Induction Step*: *To show optimality of* S_{w+1}.
  Let v be the victim chosen to make space for a_{w+1}. Therefore,

$$\begin{aligned} C_{w+1} &= C_w + a_{w+1} - v. \\ \text{Since,} \quad S_{w+1} &= \sum_{i \notin C_{w+1}} p_i.l_i, \quad \text{(by Eq. A.2)} \qquad \text{(A.3)} \\ &= S_w - p_{a_{w+1}}.l_{a_{w+1}} + p_v.l_v, \\ &= B + p_v.l_v, \quad \text{where } B = S_w - p_{a_{w+1}}.l_{a_{w+1}}. \qquad \text{(A.4)} \end{aligned}$$

  Since the broadcast is regular, the inter-arrival time of each page i is given by T/x_i, where x_i is its frequency of broadcast. Thus, the average latency of a miss for page i is half the inter-arrival time, or $T/(2 * x_i)$, since the access can occur anywhere between two consecutive occurrences of i. Thus, Equation A.4 can be re-written as,

$$\begin{aligned} S_{k+1} &= B + p_v \cdot \frac{T}{2x_v} \\ &= B + C.\frac{p_v}{x_v}, \quad \text{where } C = T/2 \end{aligned} \tag{A.5}$$

The caching strategy has no control over the values of B and C and thus, they are constants. Therefore, Equation A.5 implies that the lowest cost of data access for the client is when the page v with the lowest p_v/x_v value is chosen as the victim. This, is exactly what $\mathcal{PIX}$ does. Since it always makes this optimal decision for every replacement, subject to the restrictions listed earlier, no other algorithm can provide a lower cost of access.

Q.E.D

Bibliography

[AA95] S. Acharya and R. Alonso. The computational requirements of mobile machines. In 1^{st} *IEEE International Conference on Engineering of Complex Computer Systems*, November 1995.

[AAFZ95] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data management for asymmetric communications environments. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 199–210, San Jose, California, 22–25 May 1995.

[AB86] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation mode. *ACM TOCS*, 4(4), November 1986.

[ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 15(3), September 1990.

[AFZ96a] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on Broadcast Disks. In *Proceedings of* 22^{nd} *VLDB*, September 1996.

[AFZ96b] S. Acharya, M. Franklin, and S. Zdonik. Prefetching from a Broadcast Disk. In *Proceedings of the 12th International Conference on Data Engineering*, pages 276–287, Washington - Brussels - Tokyo, February 1996. IEEE Computer Society.

[AFZ97] S. Acharya, M. Franklin, and S. Zdonik. Balancing Push and Pull for data broadcast. In *Proceedings of ACM SIGMOD*, May 1997.

[AH93] C. Antonelli and P. Honeyman. Integrating mass storage and file systems. In *Proceedings of* 12^{th} *IEEE Symposium on Mass Storage Systems*, 1993.

[Air97] AirMedia, Inc. WWW, URL, `http://www.airmedia.com/`, July 1997.

[Amm87] M. Ammar. Response time in a Teletext system: An individual user's perspective. *IEEE Transactions on Communications*, 35(11):1159–1170, 1987.

[AS92] S. Akyurek and K. Salem. Placing replicated data to reduce seek delays. In *Proceedings of USENIX File System Conference*, May 1992.

[AW85] M. Ammar and J. Wong. The design of Teletext broadcast cycles. *Performance Evaluation*, 5, November 1985.

[AW87] M. Ammar and J. Wong. On the optimality of cyclic transmission in Teletext systems. *IEEE Transactions on Communications*, 35(1):68–73, January 1987.

[Bac97] BackWeb, Inc. WWW, URL, `http://www.backweb.com/`, July 1997.

[BAI93] B. R. Badrinath, A. Acharya, and T. Imielinski. Impact of mobility on distributed computations. *Operating Systems Review*, 27(2):15–20, 1993.

[BB97] S. Baruah and A. Bestavros. Pinwheel scheduling for fault-tolerant Broadcast Disks in real-time database systems. In *Proceedings of IEEE International Conference on Data Engineering*, Birmingham, England, April 1997.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, B. O'Neil J. Melton, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of ACM SIGMOD*, June 1995.

[Bes96a] A. Bestavros. AIDA-based real-time fault-tolerant Broadcast Disks. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Boston, MA, May 1996.

[Bes96b] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time for distributed information systems. In *Proceedings of International Conference on Data Engineering*, March 1996.

[BGH+92] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, and A. Weinrib. The Datacycle architecture. *CACM*, 35(12), December 1992.

[BI94] D. Barbara and T. Imielinski. Sleepers and Workaholics: Caching strategies in mobile environments. In *Proceedings of ACM SIGMOD*, May 1994.

[Bos97] R. Bose. Cache management in push-based systems. Master's thesis, Brown University, Providence, RI 02912, October 1997.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Caching tradeoffs in client-server DBMS architectures. In *Proceedings of ACM SIGMOD*, June 1991.

[Chi94] T. Chiueh. Scheduling for broadcast-based file systems. In *Proceedings of MOBIDATA Workshop, Rutgers University*, 1994.

[CKV93] K. Curewitz, P. Krishnan, and J. Vitter. Practical prefetching via data compression. In *Proceedings of ACM SIGMOD*, May 1993.

[CL95] P. Cao and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of ACM SIGMETRICS*, 1995.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Com95] D. E. Comer. *Internetworking with TCP/IP*, volume I. Prentice Hall, 3^{rd} edition, 1995.

[Cui97] J. Cui. Client-server performance evaluation in push-based systems. Master's thesis, Brown University, Providence, RI 02912, October 1997.

[DDY90] A. Dan, D. Dias, and P. Yu. The effect of skewed access on buffer hits and data contention in a data sharing environment. In *Proceedings of VLDB Conference*, August 1990.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[Dir96] Hughes network systems, direcpc home page. WWW, URL:: `http://www.direcpc.com/`, October 1996.

[DR92] A. Delis and N. Roussopoulos. Performance and scalability of client-server database architectures. In *International Conference On Very Large Data Bases*, pages 610–624, San Mateo, Ca., USA, August 1992. Morgan Kaufmann Publishers, Inc.

[FC92] M. Franklin and M. Carey. Client-server caching revisited. In *Proceedings of the International Workshop on Distributed Object Management*. August, 1992. Published as *Distributed Object Management*, Ozsu, Dayal, Vaduriez, eds.,Morgan Kaufmann, San Mateo, CA, 1994.

[FCL93] M. J. Franklin, M. J. Carey, and M. Livny. Local disk caching for client-server database systems. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Very large data bases, VLDB '93: proceedings of the 19th International Conference on Very Large Data Bases, August 24–27, 1993, Dublin, Ireland*, pages 641–654, Palo Alto, Calif., USA, 1993. Morgan Kaufmann Publishers.

[FJK96] M. Franklin, B. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 149–160, Montreal, Quebec, Canada, 4–6 June 1996.

[Fra93] M. Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin-Madison, 1993.

[Fra96] M. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems*. Kluwer Academic Publishers, Boston, MA, February 1996.

[Fra97] M. Franklin. Private Communication, May 1997.

[FZ96] M. Franklin and S. Zdonik. Dissemination-based information systems. *IEEE Data Engineering Bulletin*, 19(3), September 1996.

[GA94] J. Griffieon and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of USENIX Summer Technical Conference*, June 1994.

[Gif90] D. Gifford. Polychannel systems for mass digital communication. *Communications of the ACM*, 33(2), February 1990.

[GLB85] D. Gifford, J. Lucassen, and S. Berlin. The architecture for large scale information systems. In *Proceedings of ACM Symposium on Operating System Principles*, pages 161–170, 1985.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[GS95] J. Gwertzman and M Selzer. The case for geographical push caching. In *IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[GSE+94] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the International Conference on Management of Data*, pages 243–252, New York, NY, USA, May 1994. ACM Press.

[HGLW94] G. Herman, G. Gopal, K. Lee, and A. Weinrib. The Datacycle architecture for very high throughput database systems. In *Proceedings of ACM SIGMOD*, May 1994.

[How88] John H. Howard. An overview of the Andrew file system. In *Proceedings of the USENIX Winter Conference*, pages 23–26, Berkeley, CA, USA, January 1988. USENIX Association.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[HV97] S. Hameed and N. Vaidya. Log-time algorithms for scheduling single and multiple channel data broadcast. In *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM)*, September 1997.

[IB94] T. Imielinski and B. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10), October 1994.

[IV94] T. Imielinski and S. Viswanathan. Adaptive wireless information systems. In *Proceedings of SIGDBS Conference*, Tokyo, October 1994.

[IVB94a] T. Imielinski, S. Viswanathan, and B. Badrinath. Energy efficient indexing on air. In *SIGMOD*, May 1994.

[IVB94b] T. Imielinski, S. Viswanathan, and B. Badrinath. Power efficient filtering of data on air. In *Proceedings of EDBT Conference*, 1994.

[JBEA95] J. Jing, O. Bukhres, A. Elmargarmid, and R. Alonso. Bit-sequences: A new cache invalidation method in mobile environments. Technical Report CSD-TR-94-074, Computer Sciences Department, Purdue University, May 1995.

[JS94] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of VLDB Conference*, 1994.

[JW95] Ravi Jain and John Werth. Airdisks and AirRAID: Modeling and scheduling periodic wireless data. *Computer architecture news*, 23(4):23–28, September 1995.

[KE91] D. Kotz and C. Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of* 1^{st} *PDIS*, Miami Beach, FL, December 1991.

[Knu81] D. Knuth. *The Art of Computer Programming, Vol II.* Addison Wesley, 1981.

[Kor95] H. Korth. The double life of the transaction abstraction: Fundamental principle and evolving system concept. In *Proceedings of VLDB Conference*, September 1995.

[KPR94] Geoffrey H. Kuenning, Gerald J. Popek, and Peter L. Reiher. An analysis of trace data for predictive file caching in mobile computing. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 291–303, Berkeley, CA, USA, Summer 1994. USENIX.

[KS92] J. Kistler and M. Satyanarayanan. Disconnected operation in Coda file system. *ACM Transactions of Computer Systems*, 10(1):3–25, February 1992.

[KTP+96] T. Kimbrel, A. Tomkins, R. Hugo Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 19–34. USENIX Association, October 1996.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the* 8^{th} *International Conference of Distributed Information Systems*, June 1988.

[LS90] E. Levy and A. Silbershatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), December 1990.

[Mar97] Marimba, Inc. URL:: `http://www.marimba.com/`, July 1997.

[Max97] MAXIMIZED Software Home Page. WWW, URL: `http://maximized.com/`, May 1997.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 22(4), 1991.

[NW997] Internet Domain Survey. Network Wizards, Inc., WWW, URL: `http://www.nw.com/`, January 1997.

[NWO88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the SPRITE network file system. *ACM Transactions on Computer Systems, ; ACM CR 8903-0139*, 6(1), February 1988. Also published in/as: PhD Th. in progress, UCB, rcvd Mar.1988.

[OOW93] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-k page replacement algorithm for database disk buffering. In *Proceedings of ACM SIGMOD*, May 1993.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. In *Proceedings of* 14^{th} *SOSP*, December 1993.

[PG94] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Parallel and Distributed Information Systems*, pages 7–16, Los Alamitos, Ca., USA, September 1994. IEEE Computer Society Press.

[PGG+95] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: An asychronous approach. In *Proceedings of ACM SIGMOD*, page 377, Denver, CO, May 1991.

[Poi97] Pointcast, inc. WWW, URL, `http://www.pointcast.com/`, July 1997.

[PZ91] M. Palmer and S. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of VLDB Conference*, 1991.

[Qua97] Quahog Home Page, Brown University. WWW, URL: `http://www.cs.brown.edu/software/quahog`, May 1997.

[RD90] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of ACM SIGMETRICS*, Boulder, CO, 1990.

[RD91] N. Roussopoulos and A. Delis. Modern client-server DBMS architectures. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(3):52–61, September 1991.

[Sar95] Sunita Sarawagi. Query processing in tertiary memory databases. In *Proceedings of VLDB Conference*, pages 585–596, 1995.

[Sch86] H. Schwetman. CSIM: A C-based process oriented simulation language. In *Proceedings of 1986 Winter Simulation Conference*, 1986.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 3^{rd} edition, January 1997.

[SL94] S. Shekhar and D. Liu. Genesis and Advanced Traveler Information Systems (ATIS): Killer applications for mobile computing. In *Proceedings of MOBIDATA Workshop, Rutgers University*, 1994.

[SL96] S. Shekhar and D. Liu. Genesis: An approach to data dissemination in Advanced Traveller Information Systems. *IEEE Data Engineering Bulletin*, 19(3), September 1996.

[SPG91] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts, 3rd Edition*. Adison Wesley Publishing Company, 1991.

[SRB96] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcasting using Air-Cache. In *1st International Workshop on Satellite-based Information Services*, pages 30–37, November 1996.

[SRB97] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, September 1997.

[ST97] C. Su and L. Tassiulas. Broadcast scheduling for information distribution. In *Proceedings of IEEE INFOCOM*, Los Alamitos, CA, USA, April 1997. IEEE Computer Society Press.

[Ste94] W. R. Stevens. *TCP/IP Illustrated, Volume 1, The Protocols*. Addison Wesley, 1994.

[Sto90] M. Stonebraker. Architecture of future database systems. *IEEE Data Engineering Bulletin*, 13(4), December 1990.

[Tan92] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc., 1992.

[TC97] L. Tassiulas and C.Su. Optimal memory management strategies for a mobile user in a broadcast data delivery system. *IEEE Journal on Selected Areas in Communications*, 15(7), September 1997.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In M. Stonebraker, editor, *Proceedings of ACM SIGMOD*, volume 21, pages 321–330, San Diego, California, June 1992. acm, Acm Press.

[TLAC95] C. Tait, H. Lei, S. Acharya, and H. Chang. Intelligent file hoarding for mobile computers. In *Proceedings ACM Conference on Mobile Computing and Networking (Mobicom)*, November 1995.

[TX96] K. Tan and J. Xu. Energy efficient filtering of nonuniform broadcast. In *Proceedings of the* 16^{th} *International Conference in Distributed Computing Systems*, pages 520–527, 1996.

[VH96] N. Vaidya and S. Hameed. Data broadcast in asymmetric wireless environments. In *First International Workshop on Satellite-based Information Services (WOSBIS)*, November 1996.

[VI94] S. Vishwanath and T. Imielinski. Pyramid broadcasting for Video on Demand service. Technical Report DCS TR-311, Rutgers University, 1994.

[Vis94] S. Viswanathan. *Publishing in Wireless and Wireline Environments*. PhD thesis, Rutgers University, November 1994.

[WA85] J. Wong and M. Ammar. Analysis of broadcast delivery in VideoText systems. *IEE Transactions on Communication*, C-34(9):863–866, September 1985.

[WD88] J. Wong and H. Dykeman. Architecture and performance of larg scale information delivery networks. In *Proceedings of* 12^{th} *International Teletraffic Congress*, 1988.

[WN90] W. Wilkinson and M. Neimat. Maintaining consistency of client cached data. In *VLDB*, August 1990.

[Won88] J. Wong. Broadcast delivery. *Proceedings of the IEEE*, 76(12), December 1988.

[WYC96] K. Wu, P. Yu, and M. Chen. Energy-efficient caching for wireless mobile computing. In *Proceedings of ICDE*, February 1996.

[YGM95] T. Yan and H. Garcia-Molina. SIFT – a tool for wide-area information dissemination. In *Proc. 1995 USENIX Technical Conference*, 1995.

[ZFAA94] S. Zdonik, M. Franklin, R. Alonso, and S. Acharya. Are 'Disks in the Air' just Pie in the Sky? In *IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.

[ZM90] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In Zdonik and Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.