Appears in *Proceedings of ACM SIGMOD Conference*, Tuscon, Arizona, May 1997

Balancing Push and Pull for Data Broadcast*

Swarup Acharya
Brown University
sa@cs.brown.edu

Michael Franklin
University of Maryland
franklin@cs.umd.edu

Stanley Zdonik
Brown University
sbz@cs.brown.edu

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 employ broadcast to deliver data to very large client populations. We have proposed the Broadcast Disks paradigm [Zdon94, Acha95b] for organizing the contents of a data broadcast program and for managing client resources in response to such a program. Our previous work on Broadcast Disks focused exclusively on the "push-based" approach, where data is sent out on the broadcast channel according to a periodic schedule, in anticipation of client requests. In this paper, we study how to augment the push-only model with a "pull-based" approach of using a backchannel to allow clients to send explicit requests for data to the server. We analyze the scalability and performance of a broadcast-based system that integrates push and pull and study the impact of this integration on both the steady state and warm-up performance of clients. Our results show that a client backchannel can provide significant performance improvement in the broadcast environment, but that unconstrained use of the backchannel can result in scalability problems due to server saturation. We propose and investigate a set of three techniques that can delay the onset of saturation and thus, enhance the performance and scalability of the system.

1 Introduction

1.1 Data Dissemination

Many emerging applications involve the dissemination of data to large populations of clients. Examples of such *dissemination-based* applications include information feeds (e.g., stock and sports tickers or news wires), traffic information systems, electronic newsletters, software distribution, and entertainment delivery. A key enabling technology for dissemination-based applications has been the development and increasing availability of mechanisms for high-bandwidth data delivery. In particular, advances in internetworking, wireless communications, and satellite, cable, and telephone networks have enabled sophisticated content to be provided to users at the office, at home, and even on the road.

Data broadcast technology stands to play a primary role in dissemination-based applications for two reasons. First, data dissemination is inherently a *1-to-n* (or *m-to-n* where $m \ll n$) process. That is, data is distributed from a small number of sources to a much larger number of clients that have overlapping interests. Thus, any particular data item is likely to be distributed to many clients. Second, much of the communication technology that has enabled large-scale dissemination supports broadcast, and is, in some cases, intended primarily for broadcast use. For example, direct broadcast satellite providers such as Hughes' DirecPC [Dire96], and cable television (CATV) companies (through the use of high-bandwidth cable modems) are now, or will soon be, capable of supporting multi-megabit per second data broadcast to millions of homes and offices. Wireless networking technology for supporting mobile users also necessarily employs a broadcast component, and broadcast (at least in a limited form) is even supported on the Internet through protocols such as IP Multicast which is the basis of the MBone[Erik94].

1.2 Communications Asymmetry

A key aspect of dissemination-based systems is their inherent communications *asymmetry*. By communications asymmetry, we mean that the volume of data transmitted in the downstream direction (i.e., from server(s) to clients) is much greater than the volume transmitted in the upstream direction. Such asymmetry can result from several factors, including:

Network Asymmetry — In many networks the bandwidth of the communication channel from the server to

*This work has been partially supported by the NSF under grant IRI-9501353, ONR grant number N00014-91-J-4085 under ARPA order number 8220, an IBM Cooperative Graduate Fellowship, and by research funding and equipment from Intel Corporation.

the clients is much greater than the bandwidth in the opposite direction. An environment in which clients have no backchannel is an extreme example of network asymmetry. Less extreme examples include wireless networks with high-speed downlinks but slow (e.g., cellular) uplinks, and cable television networks, where bandwidth into the home is on the order of megabits per second, while a small number of slow channels (on the order of tens of kilobits per second) connect a neighborhood to the cable head.

Client to Server Ratio — A system with a small number of servers and a much larger number of clients (as is the case in dissemination-based applications) also results in asymmetry because the server message and data processing capacity must be divided among all of the clients. Thus, clients must be careful to avoid "swamping" the server with too many messages and/or too much data.

Data Volume — 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.

Updates and New Information — Finally, asymmetry can also arise in an environment where newly created items or updates to existing data items must be disseminated to clients. In such cases, there is a natural (asymmetric) flow of data in the downstream direction.

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. Thus, dissemination-based applications such as those for which the Broadcast Disks approach is intended will be asymmetric irrespective of the addition of a client backchannel.

Asymmetry imposes constraints on the behavior of clients and servers in a networked application. For example, in a system with a high client-to-server ratio, clients must limit their interactions with the server to a level which the server and backchannel can handle. World-Wide-Web (WWW) and File Transfer Protocol (FTP) servers deal with this problem by limiting the number of connections they are willing to accept at a given time. Such a limitation can result in delays when accessing popular sites, such as a site containing a new release of a popular software package or one containing up-to-date information on elections, sporting events, stocks, etc.

1.3 Push vs. Pull

In previous work we have proposed and studied Broadcast Disks as a means for coping with communications asymmetry [Zdon94, Acha95b]. This approach addresses broadcast scheduling and client-side storage management policies for the *periodic* broadcast of data. With Broadcast Disks, a 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 "broadcast 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. This approach is referred to as *push-based* data delivery. The combination of push-based data delivery and a broadcast medium is well suited for data dissemination due to its inherent scalability. Because clients are largely passive in a push-based system, the performance of any one client receiving data from the broadcast is not directly affected by other clients that are also monitoring the broadcast.

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 two obvious disadvantages to pull-based access. First, clients must be provided with a *backchannel* over which to send their requests to the server. In some environments, such as wireless networks and CATV, the provision of a backchannel can impose substantial additional expense. Second, the server must be interrupted continuously to deal with pull requests and can easily become a scalability bottleneck with large client populations. This latter problem can be mitigated, to some extent, by delivering pulled data items over a broadcast channel and allowing clients to "snoop" on the broadcast to obtain items requested by other clients.

1.4 Overview of the Paper

In this paper, we extend our previous work on data broadcasting by integrating a pull-based backchannel with the push-based Broadcast Disks approach. We focus on the tradeoffs in terms of performance and scalability between the push and pull approaches and investigate ways to efficiently combine the two approaches for both the steady-state and warm-up phases of operation. Client requests are facilitated by providing clients with a backchannel for sending messages to the server. While there are many ways in which the clients could use this capability (e.g., sending feedback and usage profiles), in this study we focus on the use of the backchannel to allow clients to *pull* pages that are not available in the client cache and that will not appear quickly enough in the broadcast.

The issues we address include: 1) the impact of client requests on steady-state and warm-up performance and the scalability of that performance; 2) the allocation of broadcast bandwidth to pushed and pulled pages; 3) techniques to maximize the benefit of client requests while avoiding server congestion; 4) the sensitivity of the performance to the variance in client access patterns; and 5) the impact of placing only a subset of the database on the Broadcast Disk, thereby forcing clients to pull the rest of the pages.

In this study, we model a system with multiple clients and a single server that controls the broadcast. Clients are provided with a backchannel, but the server has a bounded capacity for accepting requests from clients. In a lightly loaded system, backchannel requests are considered to be inexpensive. The limited capacity of the server, however, means that as the system approaches saturation, client requests become more likely to be dropped (i.e., ignored) by the server. Thus, the effectiveness of the backchannel for any one client depends on the level of backchannel activity created by the other clients.

This study adopts several assumptions about the environment:

1. *Independence of frontchannel and backchannel.* In our model, traffic on the backchannel in no way interferes with the bandwidth capability of the frontchannel. While this is not always the case for network technologies like Ethernet which share the channel, it is true of many others where the uplink and the downlink channels are physically different as in CATV networks and DirecPC.

2. *Broadcast program is static.* While having dynamic client profiles and therefore, dynamic broadcast programs is interesting, we reserve this problem for a future study. We do, however, study both the steady-state performance and the warm-up time for clients.

3. *Data is read only.* In previous work, we studied techniques for managing volatile data in a Broadcast Disk environment [Acha96b]. 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 temporarily ignored that issue here.

The remainder of the paper is structured as follows: In Section 2, we describe the extensions that we made to our previous work in order to combine pull-based data delivery with our existing push-based scheme. In Section 3, we sketch the approach that we took to simulating the combined push-pull environment, and in Section 4, we present the performance results. Section 5 discusses related work. Section 6 summarizes the results of the paper.

2 The Broadcast Setting

In this section we briefly sketch the Broadcast Disk approach (for more detail, see [Acha95b]) and present a high-level view of the extensions made to the approach to incorporate a pull-based backchannel.

2.1 Broadcast Disks

The Broadcast Disk paradigm is based on a cyclic broadcast of pages (or objects) and a corresponding set of client cache management techniques. In earlier work, we have demonstrated that the layout of the broadcast and the cache management scheme must be designed together in order to achieve the best performance.

Using Broadcast Disks, groups of pages (a.k.a., disks) are assigned different frequencies depending on their probability of access. This approach allows the creation of an arbitrarily fine-grained memory hierarchy between the client and the server. Unlike most memory hierarchies whose parameters are determined by hardware, however, the shape of this memory hierarchy can be adjusted to fit the application using software. Figure 1 shows a simple broadcast program for the seven pages named a, b, c, d, e, f, and g. These pages are placed on three disks with relative spinning speeds of 4:2:1. Page a is on the fastest disk, pages b and c are on the medium speed disk, and pages d, e, f, and g are on the slowest disk.

The number of pages in one complete cycle (i.e., the period) of the broadcast is termed the *major cycle*. The example broadcast in Figure 1 has a major cycle length of 12 pages. The algorithm used by the server to generate the broadcast schedule requires the following inputs: the number of disks, the relative frequency of each disk and assignments of data items to the disks on which they are to be broadcast. The algorithm is described in detail in [Acha95a].

Our previous work has shown that a cache replacement algorithm that is based purely on access probabilities (e.g., LRU) can perform poorly in this environment. Better performance is achieved by using a *cost-based* replacement algorithm that takes the frequency of broadcast into account. One such algorithm that we developed is called

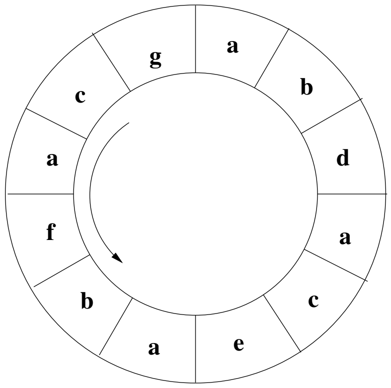


Figure 1: Example of a 7-page, 3-disk broadcast program

$\mathcal{PIX}$. If p is the probability of access and x is the frequency of broadcast, $\mathcal{PIX}$, ejects the cached page with the lowest value of p/x. Let p_i be the probability of access of page i and let x_i be the broadcast frequency of page i. If $p_a = 0.3$, $x_a = 4$, $p_b = 0.1$ and $x_b = 1$, then page a will always be ejected before page b even though its probability of access is higher. Intuitively, the value of a page depends not only on the access probability but also on how quickly it arrives on the broadcast.

2.2 Integrating a Backchannel

In this paper, we introduce a backchannel into the Broadcast Disk environment. 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 send out pages in response to client requests (as described below), and moreover, it has a finite queue in which to hold outstanding requests. If a request arrives when the queue is full, that request is thrown away.[1]

In order to support both push-based and pull-based delivery, the server can interleave 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 send 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 clients 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.

2.3 Algorithms

In the remainder of this paper, we compare 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. *Pure-Push.* This method of data delivery is the Broadcast Disk mechanism sketched above. 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.

[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 [Imie94b].

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). 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 their 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 Section 4, 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 Modeling the Broadcast Environment

The results presented in this paper were obtained using a detailed model of the Broadcast Disks environment, extended to account for a backchannel. The simulator is implemented using CSIM [Schw86]. In this section we focus on extensions to the original model needed to integrate pull-based access. Details of the original model are available in [Acha95a].

[3]The case where a server disseminates only a subset of the pages is studied in Section 4.3.

CacheSize	Client cache size (in pages)
MC_ThinkTime	Time between $\mathcal{MC}$ page accesses
ThinkTimeRatio	Ratio of $\mathcal{MC}$ to $\mathcal{VC}$ think times
SteadyStatePerc	% $\mathcal{VC}$ requests filtered through cache
Noise	% workload deviation for $\mathcal{MC}$
θ	Zipf distribution parameter

Table 1: Client Parameter Description

The simulation model is shown in Figure 2. The simulated clients (represented as $\mathcal{MC}$ and $\mathcal{VC}$ as described below) access data from the broadcast channel. They filter every request through a cache (if applicable) and through the threshold algorithm (if applicable) before submitting it to the server using the backchannel. The server can broadcast a page from the broadcast schedule or as a response to a queued page request. This choice is indicated by the *Push/Pull MUX* and is based on the value of *PullBW*. We describe the client and server models in more detail in the following sections.

3.1 The Client Model

In our previous studies we modeled only a single client because in the absence of a backchannel, the performance of any single client is independent of the other clients once the broadcast program is fixed. With a backchannel, however, there is explicit contention among the clients for the use of that channel. Thus, to model a broadcast environment with a backchannel, we must account for the activity of the competing clients.

We use two client processes to simulate an arbitrarily large client population. The first process called the Measured Client ($\mathcal{MC}$), models a single client and is analogous to the client simulated in our previous work; it is this client whose performance is reported in the simulation experiments. The second process, the Virtual Client ($\mathcal{VC}$), models the *combined effect of all other clients in the system*. The parameters that describe the operation of the $\mathcal{MC}$ and $\mathcal{VC}$ are shown in Table 1.

The simulator measures performance in logical time units called *broadcast units*. A broadcast unit is the time required to broadcast a single page. The actual response time will depend on the amount of real time required to transmit a page on the broadcast channel. The client think times are also stated in broadcast units, so the relative performance of the approaches are largely independent of the bandwidth of the broadcast medium.

The $\mathcal{MC}$ has a cache that can hold *CacheSize* pages. It accesses database pages using a Zipf [Knut81] probability distribution. The Zipf distribution (with parameter θ) is

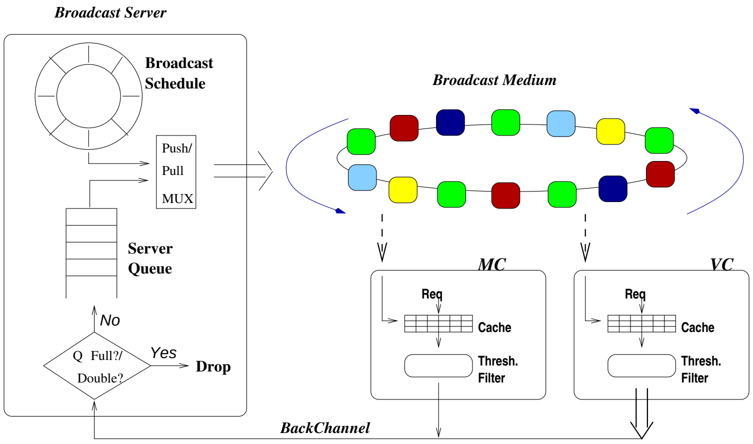


Figure 2: Simulation Model

frequently used to model skewed (i.e., non-uniform) access patterns. $\mathcal{MC}$ waits *MC_ThinkTime* broadcast units between requests. If possible, requests are serviced from the cache; otherwise, they are either satisfied by the broadcast or pulled from the server. In either case, the client sleeps until the page is retrieved.

The Virtual Client $\mathcal{VC}$, like $\mathcal{MC}$, follows a two step "request-think" loop. We use the parameter *ThinkTimeRatio* 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 broadcast, 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.

We use the parameter *SteadyStatePerc* to represent the proportion of clients in steady-state. 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*).

ServerDBSize	No. of distinct pages in broadcast
NumDisks	No. of disks
DiskSize$_i$	Size of disk i (in pages)
RelFreq$_i$	Relative broadcast frequency of disk i
ServerQSize	Size of back-channel queue
PullBW	% of broadcast slots for Pull
ThreshPerc	Threshold Factor

Table 2: Server Parameter Description

In a broadcast environment, the server generates a broadcast program taking into account the access probabilities of the all clients. Thus, it is very likely that the generated program is sub-optimal for any single client. We use a parameter called *Noise* to study the sensitivity of $\mathcal{MC}$'s response time as its access pattern diverges from that of the "average" client in the system. When *Noise*=0, clients $\mathcal{MC}$ and $\mathcal{VC}$ have exactly the same access pattern and thus, the broadcast is ideally suited for all clients. As *Noise* increases, we systematically perturb $\mathcal{MC}$'s access pattern away from $\mathcal{VC}$'s, thereby making the broadcast program less ideal for $\mathcal{MC}$. Thus, at higher values of *Noise*, $\mathcal{MC}$'s performance can be expected to quite bad. The details of how *Noise* is implemented have been described in our previous work [Acha95a].

In this study, we use the $\mathcal{PIX}$ algorithm described in Section 2.1 to manage the cache when pages are retrieved from the Broadcast Disk. Alternatively, if the clients access the database using *Pure-Pull*, the page replacement victim is chosen as the cache-resident page with the lowest probability of access (p). We call this algorithm $\mathcal{P}$.

3.2 The Server Model

The parameters that describe the operation of the server are shown in Table 2. The server broadcasts pages in the range of 1 to *ServerDBSize*. These pages are interleaved into a broadcast program as described in Section 2.1. This program is broadcast 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. $DiskSize_i$, $i \in [1..NumDisks]$, 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*. The frequency of broadcast of each disk i relative to the slowest disk is given by the parameter $RelFreq_i$. It is assumed that the lower numbered disks have higher broadcast frequency.

Since the $\mathcal{VC}$ captures the entire client population (except one client), the server uses its access pattern to generate the broadcast. The simplest strategy for assigning pages to disks is to place the $DiskSize_1$ hottest pages (from $\mathcal{VC}$'s perspective) on Disk #1, the next $DiskSize_2$ hottest pages on Disk #2, etc. This mapping, however, may be sub-optimal if clients have a cache. Heavily accessed pages tend to be cached at clients, making their frequent broadcast a waste of bandwidth. In such cases, the best broadcast program (for the steady-state) is obtained by shifting these cached pages from the fastest disk to the slowest disk. Thus, the server shifts its *CacheSize* hottest pages to the slowest disk, moving colder pages to faster disks. We call this shifted program, a broadcast with *Offset*. All results presented in this paper use *OffSet*.

The size of the server backchannel queue is given by the parameter *ServerQSize*. We assume that the client gets no feedback from the server about the success or failure of a request. A request is dropped if either the queue is already full or if there is a pre-existing queued request for the page already. The queue is serviced in a FIFO fashion and the service rate 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. As described earlier, clients make an explicit pull request only if the next arrival of the page is greater than a fixed threshold. The value of the threshold is given by *ThreshPerc* * *MajorCycleSize*, where *MajorCycleSize* is the size of a single period of the broadcast (as described in Section 2.1). Note that threshold-based filtering is not meaningful when the *Pure-Pull* approach is used. In that case, every cache miss results in an explicit request to the server, and the entire broadcast bandwidth is dedicated to satisfying client pull requests.

CacheSize	100
ThinkTime	20
ThinkTimeRatio	10, 25, 50, 100, 250
SteadyStatePerc	0%, 95%
Noise	0%, 15%, 35%
θ	0.95
ServerDBSize	1000
NumDisks	3
$DiskSize_{1,2,3}$	100, 400, 500
$RelFreq_{1,2,3}$	3, 2, 1
ServerQSize	100
PullBW	10%, 20%, 30%, 40%, 50%
ThresPerc	0%, 10%, 25%, 35%

Table 3: Parameter Settings

4 Experimental 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 is the average response time (i.e., the expected delay) at the client measured in *broadcast units*.

Table 3 shows the parameter settings for these experiments. 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. The client cache size is 100 pages (10% of the database). The back-channel queue can hold up to 100 requests for distinct pages, and the percentage of slots on the broadcast allocated to pull pages (for $\mathcal{IPP}$) is varied from 10% to 50%. 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 $\mathcal{MC}$.

Except were explicitly noted, the response time results were obtained once the client reached steady state. For the steady-state numbers, the cache warm-up effects were eliminated by starting measurements only 4000 accesses after the cache filled up and then, running the experiment until the response time stabilized. It should be noted that the results described in this section are a small subset of the results that have been obtained. These results have been chosen because they demonstrate many of the unique performance aspects and tradeoffs of this environment, and because they identify important areas for future study.

4.1 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 2.3.

4.1.1 Steady-State Performance

Figure 3(a) shows the expected average response time of requests made by the $\mathcal{MC}$ once steady-state has been reached vs. 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 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. Clients that are in the warm-up state (or that do not have caches) are likely to request the highest valued pages, 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 all clients are serviced quickly. In this case, the pull-based 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 case.

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.

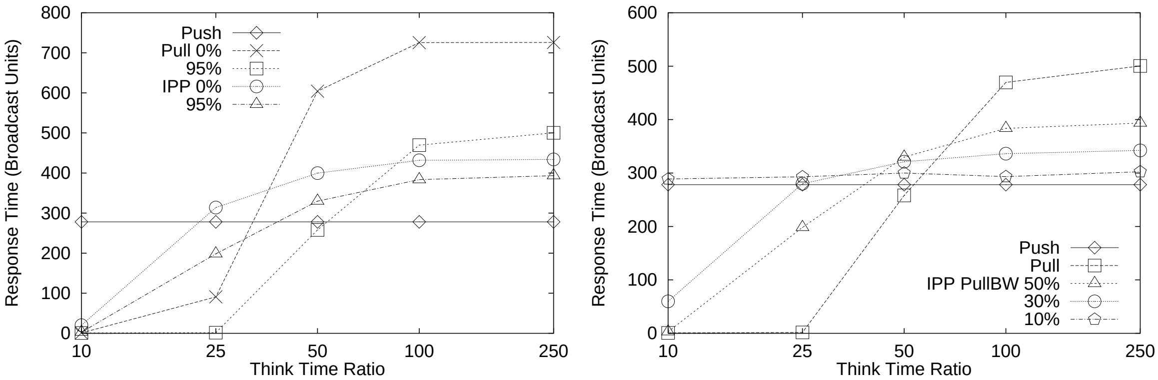
In effect, while push has a poorer response time than pull at lower loads, it provides an upper bound on the latency for any page, i.e., a "safety-net", when the server congestion causes pull requests to be dropped. This is the fundamental tradeoff between push and pull in the Broadcast Disk environment.

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

[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 4.1.4.

[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.



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

Figure 3: Steady State Client Performance

good response times over a wide range of system loads. The $\mathcal{IPP}$ algorithm is one such technique and as Figure 3 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 section and the next two experiments, 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.

4.1.2 Impact of Pull Bandwidth

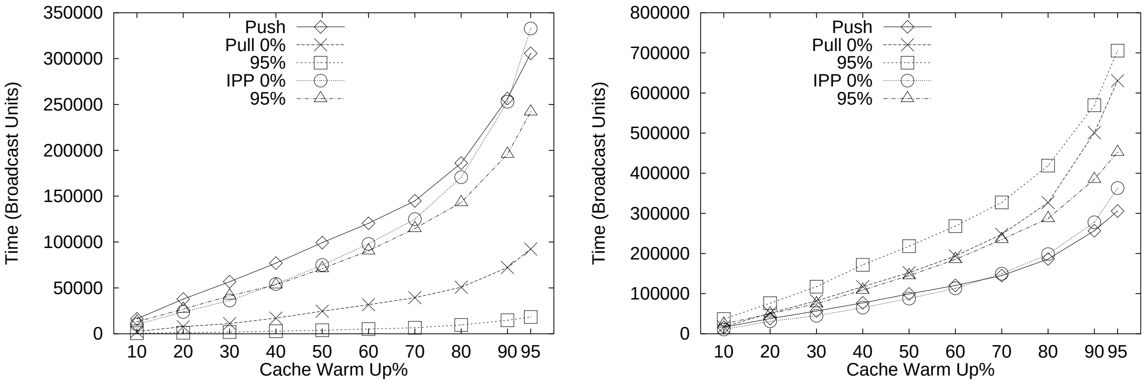
In the preceding graph, one factor driving the relative performance of the $\mathcal{IPP}$ approach compared to the two "pure" approaches was the *PullBW*, that is, the maximum percentage of the bandwidth allocated for servicing pull requests. In theory, this parameter allows $\mathcal{IPP}$ to vary between the performance of *Pure-Push* (*PullBW* = 0%) and *Pure-Pull* (*PullBW* = 100%). Figure 3(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 3(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 Disk takes 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%.

4.1.3 Warm-Up Performance

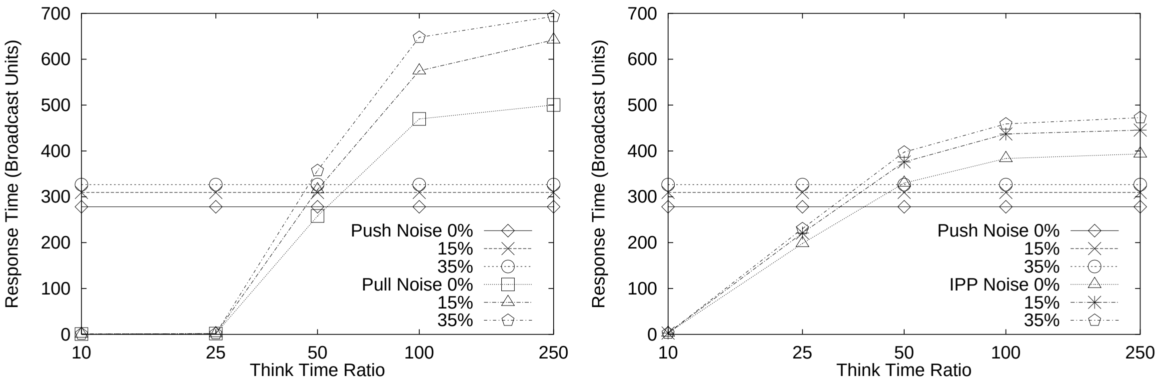
In the first part of this experiment, we investigated the steady-state performance of the $\mathcal{MC}$ under the various approaches. In this section we address the impact of the three approaches on how long it takes a client that joins the system with an empty cache to warm-up its cache. This metric is particularly important for applications in which clients are not expected to monitor the broadcast continuously. For example, in an Advanced Traveler Information System [Shek96], motorists join the "system" when they drive within range of the information broadcast.

Figure 3(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 4(a) and 4(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 3(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 4(a), under low-moderate loads, *Pure-Pull* allows the fastest warm up, with the other approaches doing significantly worse.



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

Figure 4: Client Cache Warm Up Time, $\mathcal{IPP}$ *PullBW* = 50%



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

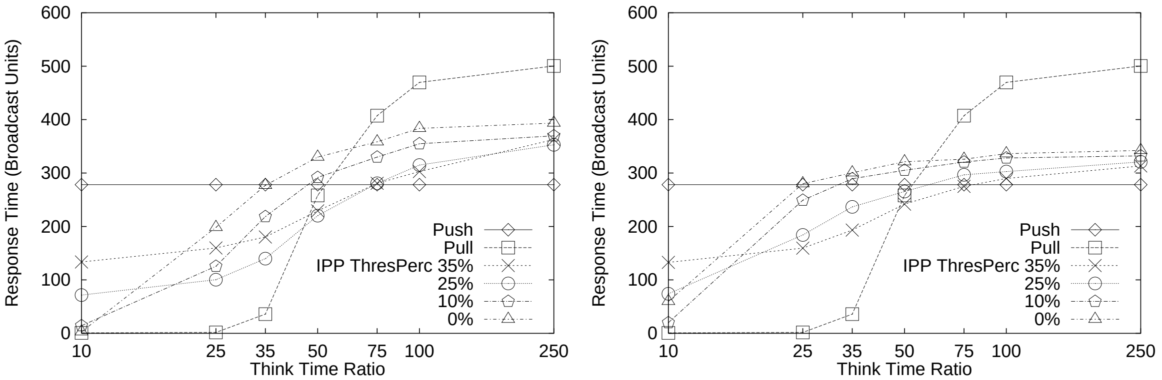
Figure 5: Noise Sensitivity, $\mathcal{IPP}$ *PullBW* = 50%

For the higher load case shown in Figure 4(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 paper, all performance results are shown for the steady-state case (i.e., *SteadyStatePerc*=95%).

4.1.4 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 described in Section 3. 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 $\mathcal{VC}$ (whose access pattern 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 5(a) and 5(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 5(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,



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

Figure 6: Influence of Threshold on Response Time

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 5(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.

4.2 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 3(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 2.3, 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 vs. backchannel usage and server congestion.

[7] Because 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.

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 6(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 3(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 6(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 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 system 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.

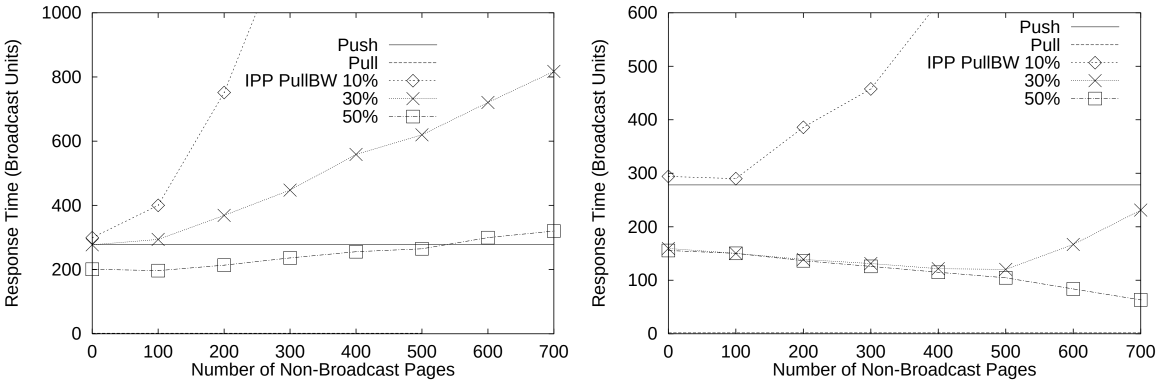
[8]Because of the shape of the multi-disk used to structure the push schedule, approximately half of the pages appear at least once in the first 25% of the broadcast period here.

4.3 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 4.1.2, 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. As is discussed in Section 5, a similar approach was advocated in [Imie94c, Vish94]. 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.

Figure 7 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. 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 7(a) and 7(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



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

Figure 7: Restricting Push Contents, *ThinkTimeRatio* = 25

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 7(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 7(b) shows the performance for the same simulation settings as Figure 7(a), except that *ThresPerc* is set to 35%. Since clients filter some of their requests due to threshold, 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 Experiments 1 and 2, the response time is *not bounded* in the worst case by *Pure-Pull*. This is seen in Figure 8 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 7 of the need for adequate pull bandwidth to respond to client requests for non-push schedule pages is again apparent here in the crossover among the $\mathcal{IPP}$ 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 and the "safety net" of the push schedule cause the order of the $\mathcal{IPP}$ algorithms to invert on the right-hand side of the graph from the ordering on the lightly loaded left-hand side.

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 the entire range of server load.

[9] The line for *Pure-Pull* is not visible in Figures 7(a) and 7(b) because of overlap with the x-axis.

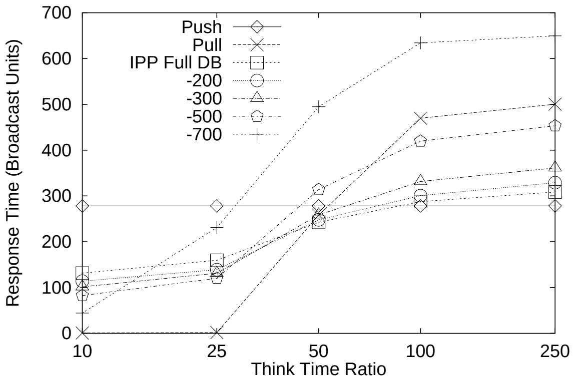


Figure 8: Server Load Sensitivity for Restricted Push
PullBW = 30%, *ThresPerc* = 35%

4.4 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, a merge between the two extremes of *Pure-Pull* and *Pure-Push* to provide reasonably 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 achieve the goal of delaying 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 that we described for avoiding saturation is one in which successively larger pieces are chopped off 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. 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 scheduled broadcast.

5 Related Work

The basic idea of managing broadcast data has been investigated by a number of projects [Amma85, Herm87, Giff90, Bowe92, Imie94a, Imie94b]. Our work on Broadcast Disks differs from these in that we consider multi-level disks and their relationship to cache management. In [Acha95a], we proposed an algorithm to generate Broadcast Disk programs and demonstrated the need for cost-based caching in this environment. In [Acha96a], we showed how opportunistic prefetching by the client can significantly improve performance over demand-driven caching. More recently, in [Acha96b], we studied the influence of volatile data on the client performance and showed that the Broadcast Disk environment is very robust in the presence of updates.

The Datacycle Project [Bowe92, Herm87] at Bellcore investigated the notion of using a repetitive broadcast medium for database storage and query processing. An early effort in information broadcasting, the Boston Community Information System (BCIS) is described in [Giff90]. BCIS broadcast news articles and information over an FM channel to clients with personal computers specially equipped with radio receivers. Both Datacycle and BCIS used a flat disk approach. The mobility group at Rutgers [Imie94a, Imie94b] has done significant work on data broadcasting in mobile environments. A main focus of their work has been to investigate novel ways of indexing in order to reduce power consumption at the mobile clients. Some recent applications of dissemination-based systems include information dissemination on the Internet [Yan95, Best96], Advanced Traveler Information Systems [Shek96] and dissemination using satellite networks [Dao96].

Work by Imielinski and Viswanathan [Imie94c, Vish94] is more closely related to the results we report here, 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 4.3), 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 key 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 [Imie94c, Vish94], only pages in the "on-demand" group can be requested over the back channel. 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.

There has also been work on broadcasting in Teletex systems [Amma85, Wong88]. [Wong88] presents an overview of some of the analytical studies on one-way, two-way and hybrid broadcast in this framework. However, as in [Imie94c, Vish94], their server models are significantly different from ours.

6 Conclusions

In this paper, we addressed the problem of adding a backchannel to a broadcast-based environment. We showed that in highly asymmetric cases, the server can become a bottleneck and using the backchannel to simply pull all misses is not a good strategy. We showed how a few simple techniques can be used to improve this situation. 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.

We believe that this paper carefully delineates the issues and the tradeoffs that one must confront in this type of environment. We also believe that we have developed a realistic simulation model that was used to advantage in this study and can be used in future studies. This model is based on a finite server queue and a fixed service time (as reflected in *PullBW*). It allows us to mix push and pull in various ways to study their relative merits. The model also supports the incorporation of client caching.

Beyond what was presented, we would like to develop tools to make the parameter setting decisions for real dissemination-based information systems easier. These tools could be analytic or they could be a better interface on our simulator. To this end, we would be interested in adapting the analytical framework presented in [Imie94c, Wong88] to develop solutions for our model. We also see the utility in developing more dynamic algorithms that can adjust to changes in the system load. For example, as the contention on the server increases, a dynamic algorithm might automatically reduce the pull bandwidth at the server and also use a larger threshold at the client. We are also in the process of implementing a Broadcast Disk facility that will include support for a backchannel. This will allow us and others to easily implement real applications and further test our current results.

Finally, as stated in [Fran96], the periodic push approach of Broadcast Disks and the pull-based approach of request-response are just two of many ways for delivering data to clients. We see the integration of these two techniques as described in this paper as a first step towards the development of the more general notion of a *dissemination-based information system* (DBIS). A DBIS is a distributed information system in which it is possible to freely mix data delivery options. For example, the nodes in a DBIS could communicate using periodic push, request-response, publish/subscribe [Oki93, Yan95, Glan96], and other techniques in various combinations.

An integrated DBIS consists of data sources, consumers (i.e., clients), and *information brokers*. These brokers acquire information from sources, add value to that information (e.g,. some additional computation or organizational structure) and then distribute this information to other brokers or clients. By creating hierarchies of brokers, information delivery can be tailored to the needs of many different users. Our long-term goal is to develop tools that would allow the different data delivery mechanisms to be mixed and matched, so that the most appropriate mechanism can be used for each type of data flow in the system.

Acknowledgments

The authors would like to thank Rafael Alonso for his ideas during the early phases of this work.

References

[Acha95a] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communication Environments", *Proc. ACM SIGMOD Conf.*, San Jose, CA, May, 1995.

[Acha95b] S. Acharya, M. Franklin, S. Zdonik, "Dissemination-based Data Delivery Using Broadcast Disks", *IEEE Personal Communications*, 2(6), December, 1995.

[Acha96a] S. Acharya, M. Franklin, S. Zdonik, "Prefetching from a Broadcast Disk", *12th International Conference on Data Engineering*, New Orleans, LA, February, 1996.

[Acha96b] S. Acharya, M. Franklin, S. Zdonik, "Disseminating Updates on Broadcast Disks", *Proc.* 22^{nd} *VLDB Conf.*, Bombay, India, September, 1996.

[Amma85] M. Ammar, J. Wong, "The Design of Teletext Broadcast Cycles", *Perf. Evaluation*, 5 (1985).

[Best96] A. Bestavros, C. Cunha, "Server-initiated Document Dissemination for the WWW", *IEEE Data Engineering Bulletin*, 19(3), September, 1996.

[Bowe92] T. Bowen, G. Gopal, G. Herman, T. Hickey, K. Lee, W. Mansfield, J. Raitz, A. Weinrib, "The Datacycle Architecture", *CACM*, 35(12), December, 1992.

[Dao96] S. Dao, B. Perry, "Information Dissemination in Hybrid Satellite/Terrestrial Networks", *IEEE Data Engineering Bulletin*, 19(3), September, 1996.

[Dire96] Hughes Network Systems, DirecPC Home Page, `http://www.direcpc.com/`, Oct, 1996.

[Erik94] H. Erikson,"MBONE: The Multicast Backbone", *CACM*, 37(8), August, 1994.

[Fran96] M. Franklin, S. Zdonik, "Dissemination-Based Information Systems", *IEEE Data Engineering Bulletin*, 19(3), September, 1996.

[Giff90] D. Gifford, "Polychannel Systems for Mass Digital Communication", *CACM*, 33(2), February, 1990.

[Glan96] D. Glance, "Multicast Support for Data Dissemination in OrbixTalk", *IEEE Data Engineering Bulletin*, 19(3), September, 1996.

[Herm87] G. Herman, G. Gopal, K. Lee, A. Weinrib, "The Datacycle Architecture for Very High Throughput Database Systems", *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May, 1987.

[Imie94a] T. Imielinski, B. Badrinath, "Mobile Wireless Computing: Challenges in Data Management", *CACM*, 37(10), October, 1994.

[Imie94b] T. Imielinski, S. Viswanathan, B. Badrinath, "Energy Efficient Indexing on Air", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May, 1994.

[Imie94c] T. Imielinski, S. Viswanathan, "Adaptive Wireless Information Systems", *Proc. of SIGDBS Conf.*, Tokyo, October, 1994.

[Knut81] D. Knuth, *The Art of Computer Programming, Vol II*, Addison Wesley, 1981.

[Oki93] B. Oki, M. Pfluegl, A. Siegel, D. Skeen, "The Information Bus - An Architecture for Extensible Distributed Systems", *Proc. 14th SOSP*, Ashville, NC, December, 1993.

[Schw86] H. D. Schwetman, "CSIM: A C-based process oriented simulation language", *Proc. 1986 Winter Simulation Conf.*, 1986.

[Shek96] S. Shekhar, A. Fetterer, D. Liu, "Genesis: An Approach to Data Dissemination in Advanced Traveller Information Systems", *IEEE Data Engineering Bulletin*, 19(3), September, 1996.

[Wong88] J. Wong, "Broadcast Delivery", *Proceedings of the IEEE*, 76(12), December, 1988.

[Vish94] S. Viswanathan, "Publishing in Wireless and Wireline Environments", *Ph.D Thesis*, Rutgers Univ. Tech. Report, November, 1994.

[Yan95] T. Yan, H. Garcia-Molina, "SIFT – A Tool for Wide-area Information Dissemination", *Proc. 1995 USENIX Technical Conference*, 1995.

[Zdon94] S. Zdonik, M. Franklin, R. Alonso, S. Acharya, "Are 'Disks in the Air' Just Pie in the Sky?", *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December, 1994.