The Multicast Pull Advantage in Dissemination-based Data Delivery

Jonathan Beaver, Kirk Pruhs, Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{beaver, panos, kirk}@cs.pitt.edu *

Vincenzo Liberatore
Division of Computer Science
Case Western Reserve University
Cleveland, Ohio 44106-7071
vincenzo.liberatore@cwru.edu †

ABSTRACT
A major problem in web database applications and on the Internet in general is the scalable delivery of data. Multicast is one of the standard techniques to achieve scalable data dissemination. However the use of multicast introduces a variety of data management issues at the server. In this paper we examine three major problems namely, the push popularity problem, the document classification problem and the bandwidth division problem, that arise in the design of a hybrid data dissemination scheme. We propose solutions to these problems and argue that these are essentially the best possible solutions. In particular, we argue for having a multicast pull mode, in addition to the traditional unicast pull mode and the commonly proposed multicast push mode. We give a simple method for estimating the current popularities of pushed documents. We give an algorithm for determining which documents should be pushed/pulled, and for determining how much of the server bandwidth should be devoted to push/pull. We report on experiments with our system that validate our algorithms.

1. INTRODUCTION
It is indisputable that the web has brought massive amounts of information to everyone's fingertips, changing forever the way we learn the news, perform research, do business, deal with disasters, etc. It is also indisputable that a major problem in these applications is the scalable delivery of data. This problem is particularly acute exactly at the time when the scalability of data delivery is most important. Examples that one can find cited in the popular press include: news about the terrorist attacks at `msnbc.com` during 9/11/2001, virus patches from `mcafee.com` during the recent Slammer virus, and weather reports at the Federal Emergency Management Agency `fema.gov` during hurricane Isabel around September 18, 2003.

Several front-end and back-end web server techniques have been proposed to address this scalability problem. These are predominantly data caching and replication methods; the front-end ones are along the client/web server communication path and include proxy caches [14] and server-side caches [13] whereas the back-end techniques are along the path from the web-server and database servers and include web server cache plug-in mechanisms and asynchronous caches [20, 21, 6, 22]. However, the scalability gains that these approaches can offer are constrained by the underlying traditional *unicast pull*, where data are delivered from servers to clients on demand.

The experience with the use of broadcasting as a means of disseminating information to large client populations in wireless settings has led to the realization that a wide range of emerging web database applications can also benefit from a (broadcast) *multicast push* mode of data dissemination. The literature to date suggests that multicast be reserved solely for hot/popular pushed documents [1, 17]. In this paper, we argue that data can be more efficiently delivered to a large number of users using a hybrid dissemination scheme that includes a *multicast pull* mode in addition to the multicast push and the traditional unicast modes. To test our hypothesis, we have built a prototype middleware that supports such a hybrid dissemination scheme and acts as a reverse-proxy to a Web server for the delivery of documents which are materialized views. The prototype has allowed us to investigate the interaction of these dissemination modes, and to build the argument in favor of our proposed hybrid scheme by answering in sequence each one of the following questions.

First, we need to establish if a *multicast pull* channel should even be present. Our server[1] uses the multicast pull when it receives near simultaneous requests for the same not-hot document. It has been observed that in practice the document popularities satisfy a Zipf distribution [8]. We show experimentally that if document popularities have a *static* Zipf distribution, then a multicast pull channel gives a modest, but noticeable, improvement in average response. We then consider the case of dynamic distributions with moving hot spots. That is, the popularity distribution remains Zipf, but the identity of the popular documents change over time. For example, consider a cricket web site. The biography page for a particular player may suddenly become very popular after he makes a particularly good play (say bowling a hat-trick), or a particularly bad play (say an out hit wicket). We show that multicast pull significantly improves average response time during the transition period when the server is adjusting to this change in document popularities.

*Supported in part by NSF grant CCR-0098752, NSF grant ANI-0123705, and NSF grant ANI-0325353
†Supported in part by NSF grant ANI-0123929.

[1]In the rest of the paper, the term server refers to both the prototype middleware and the web/database back-end servers.

The second question pertains to the estimation of current popularity of pushed documents. We call this the *push popularity* problem. The rationale for investigating the push popularity problem is that documents are assigned to dissemination modes depending on their popularity. If a pushed document becomes sufficiently less popular then it should no longer be pushed. One solution for the push popularity problem proposed in [29] was to occasionally drop each pushed document from the push channel, thus forcing clients to send explicit requests. This solution has the disadvantage that, if the document is very popular, then the server might be temporarily flooded by requests for this document. We propose that the server publish a report probability for each pushed document. Then when a client receives a user request for a pushed document, it submits an explicit request for that document with probability equal to this report probability. The report probabilities should be small enough that server is almost surely not going to be overwhelmed with requests for pushed documents. We show that if the goal is to minimize the maximum relative inaccuracy observed in the estimated popularities of the pushed documents, then each report probability should be set inversely proportional to the estimated access probability for that document. We believe that our solution is both a more scalable, and more straight-forward, solution to the problem of estimating the popularity of pushed documents.

Having established the necessity of multicast pull and an effective way to estimate push popularity, we can proceed with the assignment of documents and the allocation of bandwidth to each dissemination mode. In our terminology, the *document classification* problem is the issue of determining which documents are pushed and which documents are pulled; the *bandwidth division* problem is the issue of determining how much of the the server bandwidth should be devoted to pushed documents and how much of the server bandwidth should be devoted to pulled documents when the server is loaded. We will argue that the document classification problem and the bandwidth division problem are interrelated. We solve these two problems simultaneously using a modification of an algorithm proposed in [7]. In [7], it is shown how to minimize the bandwidth given a fixed latency per request. The bandwidth division problem is the dual of this problem. That is, the bandwidth division problem asks how to minimize latency given a fixed known system bandwidth. Using the algorithm in [7] as a subroutine, we show how to simultaneously solve the document classification problem and the bandwidth division problem in linear time.

The primary contribution of this paper is that it presents a unified and rigorous analysis of hybrid data dissemination schemes involving unicast and multiple multicast modes. The specific contributions of the paper are as follows:

- The first contribution is the quantification of the advantage of including a multicast pull component.
- The second contribution is our algorithm for the push popularity problem.
- The third contribution is the essentially optimal algorithm for document classification and bandwidth division.

To the best of our knowledge, we deal with these issues in a more complete way than previous papers in the literature. Although we present these issues in the context of a wire-line environment, our future work is to look at these contributions in the context of data dissemination to mobile users in a wireless environment.

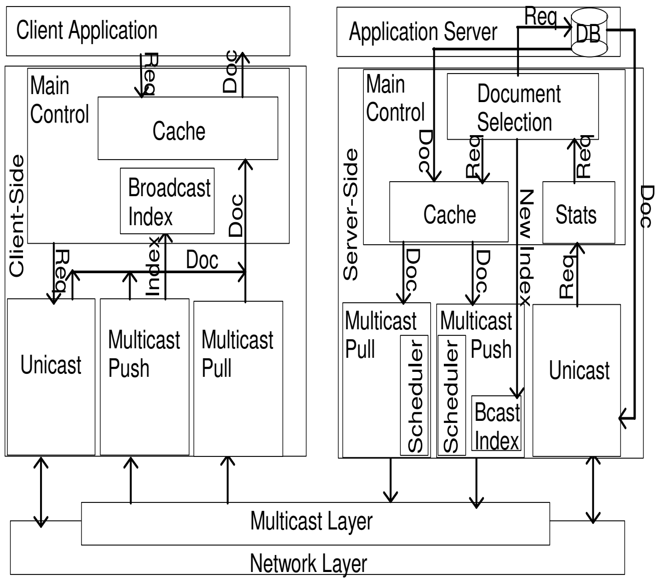

Figure 1: Our system architecture

Note that once the bandwidth division and document selection problems have been solved, one is then left with the problem of scheduling documents within the multicast push channel, the multicast pull channel, and the unicast channel. Scheduling documents within a particular channel has been well studied. In particular, in the wireless environment, the problem of scheduling within a broadcast pull channel has received much attention [2, 3, 15, 26, 27, 30]. These scheduling problems are orthogonal to our bandwidth division and document selection problems, where those problems address when to send and our work is addressing the issue of what to send.

The remaining of the paper is structures as follows: Section 2 describes our middleware server and our middleware clients which form the core of our prototype system. We also discuss the rationale behind our design choices. In Section 3, we report on experiments with our system that validate our algorithms. In section 4 we very briefly survey related work. Section 5 summarizes our observations and contributions.

2. SYSTEM DESCRIPTION

In this section, we briefly describe our middleware, whose configuration is shown in Figure 1. The configuration demonstrates the main building blocks, each tied to a particular data management function in multicast environments (document selection, scheduling, consistency, caching, and indexing). The middleware has server-side and client-side components; the client-side sits under a client application such as a browser. The server-side might either sit under a web server, or be an alternative to a web/database server. Most of the server-side building blocks have a corresponding module on the clients and vice versa.

Both the client and server sit above some multicast capable service. The middleware is connected to the multicast service through a thin Transportation Adaptation Layer (TAL). So the identity of the multicast service is not particularly relevant to our middleware. For our experiments in this paper we use Java Reliable Multicast Service (JRMS) [25]. JRMS guarantees the reliable delivery of multicast packets. We maintain two multicast channels, one for pull and one for push.

2.1 Client Description

In order to allow the clients to quickly determine whether a needed document(materialized view) is in the current hot broadcast set, the server broadcasts an index of sorted documents on the multicast push channel.

When the client receives a request from the application client for a document D_i, the client takes the following actions.

- The client first checks whether D_i is listed on its stored version of the most recent index of the documents on the multicast push channel. If D_i is in the index, the client sends an explicit request for this document to the server with the report probability s_i specified for that document in the index, and then waits for D_i to appear on the multicast push channel.
- If D_i is not in the latest stored version of the multicast push index, then the client makes a direct request for D_i to the server using HTTP/TCP.
- After making this request, the client monitors the TCP connection, and both multicast channels. When the client receives D_i, it passes the document back to the application client.

The client must monitor the multicast push channel even if the document was not found in the push index because it it possible that the server has just started to push the document. If client did not monitor the multicast push channel, we could have a race condition.

The server may close the TCP connection if it determines that it will not serve the document over this TCP connection. In case that the client sees the TCP connection being closed, it knows that the server intends to send the document on one of the multicast channels, and the client just continues to monitor the multicast channels.

2.2 Server Description

When the server receives a request for a document D_i from the client over a TCP connection, one of three actions can be taken.

- If the requested document is currently on the push queue, the count of recent access to D_i is incremented by $1/s_i$. Recall s_i is the probability that a client reports an access to D_i. So on average this request represents $1/s_i$ unreported requests. Further the server closes this TCP connection as it will serve this document over the multicast push channel.
- If D_i is not on the push channel then the count of recent access to D_i is incremented by 1. If a request for D_i is already on the server's out going queue of pulled documents, then the count of the current outstanding requests for D_i is incremented. If the outstanding request count for D_i is over the predefined multicast pull threshold then all TCP connections for this document are closed. The server can close these TCP connections because it knows that it will transmit D_i on the multicast pull channel.
- If a request for D_i is not on the outgoing queue of pulled documents, then this request is enqueued on the pull queue with the outstanding request count initialized to 1.

There is a thread that dequeues documents from the push and pull queues and transmits the documents. The documents on the push channel are broadcast using a flat broadcast, that is each document on the push channel is pushed with equal frequency. The fraction of time that this thread chooses from each queue is specified by the bandwidth division algorithm. When a request for D_i reaches the front of the outgoing pull queue, its request count is compared to the multicast pull threshold. If the count is above this threshold, then the document is transmitted on the multicast pull channel. Otherwise, the request is transmitted over the open TCP channels. All queues are processed in a FIFO manner.

Asynchronously the server has a thread to occasionally solve the document classification problem, the bandwidth division problem, and computes the report probabilities for the pull documents. We now describe in more detail how this is accomplished.

2.2.1 Document Classification and Bandwidth Division

To understand the algorithm for document classification and bandwidth division, it is first necessary to understand the different nature of average latency for the multicast push and for the unicast pull channel. The average latency for documents on the multicast push channel is roughly linear in the number of documents on this channel. Specifically, the average latency for a document on the multicast push channel is half of the period of the broadcast cycle since we assume that documents are broadcast sequentially. Note that this assumes that the time to broadcast a single document is negligible with respect to the period of the broadcast cycle. However, the delays for pulled documents are radically different from those of pushed documents. If document i is assigned to unicast pull, a client request for i is queued at the server for transmission. Let S_i be the size of document i. Basic queuing theory tells us that the corresponding queuing delay is either $O(S_i)$ or unbounded, depending on whether the server load is less than 1 or not. Thus, to minimize average latency, the server should require as many documents as possible be pulled, as long as the load for the pulled documents is bounded by a constant less than 1.

Our solution to document classification and bandwidth division is to use an integrated algorithm that minimizes average latency. The starting point is a method suggested by [7] that minimizes the bandwidth B to achieve a target latency L. The known method is not directly applicable to document classification and bandwidth division because our goal, on the contrary, is to minimize the latency L given a fixed amount of available server bandwidth B. However, the previous method will be useful as a component of the final algorithm.

For the purpose of analysis, client requests follow a Poisson process in which document i is requested with probability p_i and the aggregate request rate is λ. The popularity profile p_i and the rate λ can be determined by the push popularity algorithm described in Section 2.3. Algorithm 1 uses a target average latency L. Its goal is, as in [7], to partition the documents between unicast pull and multicast push and to split the system bandwidth so as to minimize the system bandwidth B required to achieve latency L. If document i is assigned to the pull channel, it will use bandwidth $\lambda p_i S_i$. If document i is assigned to the push channel, it will use bandwidth S_i/L, which is also the rate at which the document must be broadcast to give *worst-case* response time L. It was then stated [7] that a document should be pushed if

$$\lambda p_i S_i > \frac{S_i}{L} . \qquad (1)$$

Algorithm 1 follows this approach. However, as we are interested in the *average* latency of a pushed document instead of the worst-case latency, we need to make the following modification to equation (1). The unicast pull term $\lambda p_i S_i$ in (1) is the bandwidth required to obtain average latency L, and thus the multicast push bandwidth also needs to be the bandwidth required to achieve an average latency of L for the comparison to be meaningful. In this case, the bandwidth required by this document on the push channel to achieve average-case latency L is $S_i/(2L)$. Thus a document should be pushed if $\lambda p_i S_i > S_i/(2L)$. The resulting subroutine is shown below as Algorithm 1 and its parameters are summarized in Table 1.

Parameter	Description
n	number of documents
λ	observed request rate λ
α	pull over-provisioning factor
L	current required latency
B	total available system bandwidth
S	array of document sizes S_i
p	array of document probabilities p_i
ϵ	tolerance factor

Table 1: Parameters for Algorithms 1 and 2.

Algorithm 1 tryLatency

```
Require: $n, \lambda, L, p$ as defined in Table 1
Ensure: Returns the number $k$ of items pushed given that average
    latency of $L$ is required
1: while max − min > 1 do
2:     $k \leftarrow (\max + \min)/2$
3:     if $(p_k \lambda L) > 1/2$ then
4:         min ← $k$
5:     else
6:         max ← $k$
7:     end if
8: end while
9: Return $k$
```

The bandwidth for the push channel is now $\sum_{i:p_i>1/(\lambda L)} S_i/(2L)$ and the bandwidth for the unicast requests is $\sum_{i:p_i\leq 1/(\lambda L)} \lambda p_i S_i$. In particular, if L increases, while all other variables remain fixed, then more documents are pushed, an observation that will be used to derive the final algorithm.

The second and more substantial modification to the previous argument is due to the mismatch between the objectives of Algorithm 1 and the desired objectives for bandwidth division and document selection. Algorithm 1 minimizes the amount of required bandwidth to achieve a fixed L, whereas our goal is to minimize L given a fixed deployed bandwidth B. In some sense, our problem is the dual of the one that Algorithm 1 solves.

Algorithm 2 solves the bandwidth division and document selection problems, and uses Algorithm 1 as a subroutine. The algorithm employs a parameter $\alpha > 1$ that measures the target level of over-provisioning for the pull channel. More precisely, the actual bandwidth we reserve for pull is α times what an idealized estimate predicts. Queuing theory asserts that $\alpha > 1$ guarantees bounded queuing delays, whereas $\alpha \leq 1$ leads to infinite queuing delays. As such, the parameter α can also be thought of as a safety margin for the pull channel. The algorithm also uses a parameter $\epsilon > 0$, which is an arbitrarily small positive number, and finds a solution that has latency within ϵ of the optimum for the given bandwidth and popularities.

Algorithm 2 Bandwidth Division and Document Classification

```
Require: $n, \lambda, \alpha, B, S, p$, and $\epsilon$ as defined in Table 1, and $p_i \geq$
    $p_{i+1}$ $(1 \leq i < n)$
Ensure: $k$ is the optimal number of documents on the push chan-
    nel, pullBW is the optimal pull bandwidth, pushBW is the op-
    timal push bandwidth
 1: for $i = 1, \ldots, n$ do
 2:     $\text{rspt}_i \leftarrow \text{rspt}_{i-1} + p_i S_i \lambda$
 3:     $\text{sizeTotal}_i = \text{sizeTotal}_{i-1} + S_i$
 4: end for
 5: lMax ← $\text{sizeTotal}_n/B$
 6: lMin ← 0
 7: while lMax − lMin > $\epsilon$ do
 8:     $L \leftarrow (\text{lMax} + \text{lMin})/2$
 9:     $k \leftarrow \text{tryLatency}(L, p, \lambda, n)$
10:     pullBW ← $\alpha(\text{rspt}_n - \text{rspt}_k)$
11:     pushBW ← $B$ − pullBW
12:     if pushBW ≥ $(\text{sizeTotal}_k/(2L))$ then
13:         lMax ← $L$
14:     else
15:         lMin ← $L$
16:     end if
17: end while
```

Algorithm 1 assumes that documents have been sorted in non-increasing order of popularity, i.e., $p_i \geq p_{i+1}$ $(1 \leq i < n)$. It can be easily seen that if i is pushed, then $j < i$ should be pushed as well. Then, the problem becomes that of finding a value of k such that the multicast push set $\{1, 2, \ldots, k\}$ minimizes the latency L given a certain bandwidth B and pull over-provisioning factor α. The optimal value k^* can be found by trying all possible values of L, computing the document k that achieves L with Algorithm 1, and checking that this value of k satisfies the bandwidth requirements. The pull bandwidth requirement is $\alpha \sum_{i=k+1}^{n} \lambda p_i S_i$, which leaves pushBW $= B - \alpha \sum_{k+1}^{n} \lambda p_i S_i$ for the push channel, and average latency for the pushed documents of $\sum_{i=1}^{k} S_i/2\text{pushBW}$. If this computed average latency for the pushed documents is greater than L, then L needs to be increased, otherwise L needs to be decreased.

Algorithm 2 follows this approach but with two optimizations. In the first place, the algorithm performs a binary search over all possible values of L and stops when the interval for L is bounded by the tolerance ϵ. Moreover, the algorithm pre-computes the sums $\sum_{i=1}^{k} \lambda p_i S_i$ and $\sum_{i=1}^{k} S_i$ in the arrays rspt and the sizeTotal respectively (Lines 1–4). The purpose of these computations is to use the totals in the place of the sums in the bandwidth computations. Because of this optimization, the portion of the algorithm before the binary search runs in linear time. The maintenance of the rspt and sizeTotal arrays can be implemented in logarithmic time per query using standard augmented binary tree techniques [11]. Thus, the running time of algorithm 2 is $O(\max(n, \log(\frac{\sum_{i=1}^{n} S_i}{B\epsilon})))$. We expect that as a practical matter that the running time will be $O(n)$.

2.3 Report Probabilities

Document selection and bandwidth division rely on estimates p of document popularity. The values of p can be estimated by sampling the client population as follows. The server publishes a report probability s_i for each pushed document i. Then, if a client wishes to access document i, it submits an explicit request for that document with probability s_i. In principle, clients would not need to submit any request for push documents, but if they do send requests with probability s_i, the server can use those requests to estimate p_i. At the same time, the report probability s_i should be small enough that server is almost surely not going to be overwhelmed with requests for pushed documents. In particular, we consider the objective of minimizing the maximum relative inaccuracy observed in the estimated popularities of the pushed documents. In this case, we show analytically that each report probability should be set inversely proportional to the predicted access probability for that document.

First, the server calculates the rate β of incoming reports that it can tolerate. Presumably, β is approximately equal to the rate that the server can accept TCP connections minus the rate of connection arrivals for pulled documents. Therefore, the value of β can be estimated from the access probabilities and the current request rate, all scaled down by a safety factor to give the server a little leeway for error. Then, the s_i's have to be set such that $\sum_{i=1}^{k} \lambda p_i s_i \leq \beta$, where documents $1, \ldots k$ are on the push channel. The expected number of reports μ_i that the server can expect to see for i over a unit time period is $\lambda p_i s_i$. Using standard Chernoff bounds, the probability that number of reports is more than $(1+\delta)\mu_i$ is roughly $e^{\frac{-\mu_i \delta^2}{4}}$, and that the probability that number of reports is less than $(1-\delta)\mu_i$ is roughly $e^{\frac{-\mu_i \delta^2}{2}}$. If the goal is to minimize the expected maximum relative inaccuracy of the reports, all of the upper tail bounds should be equal and all of the lower tail bounds should be equal. That is, all μ_i should be equal, or equivalently it should be the case that for all i, $1 \leq i \leq k$, $s_i = \frac{\beta}{\lambda p_i k}$. Hence, each document should have a report percentage inversely proportional to its access probability.

3. EXPERIMENTS

The simulation environment we used in our experiments is a prototype of the system we explained in Section 2. Sitting on top of the middleware client was a simulated application client generating Poisson requests for fixed modestly sized documents with a Zipf distributed probability distribution.

We restricted our simulation to fixed sized documents for a couple reasons. One reason is that it is not clear what the right joint probability distribution is between document size and popularity [9]. Another reason is that is has been said that the correlation between object size and popularity, if any, is weak and can be ignored [8]. Finally, examination of our algorithm shows that variable sized documents do not effect the choice of which items to put on the push channel and which on the pull channels, as popularity is the main driving force.

We ran both the client and server on the same machine to limit the amount of noise that could be introduced because of the network. We implemented a background request filler that simulates a specified number of clients, and sends requests to the server. The requests by the filler are treated identically to those made by our client, except that we do not record latency for these requests. We run our experiments for 10000 requests in order to achieve statistically correct results. All the figures reflect the average results of these runs. The computer used in these simulations was a 2.0Ghz dual processor machine with 1.2MB of RAM and running Linux. JRMS was used for multicasting.

Parameter	Value	Default
Document Size	0.5K bytes	0.5K bytes
Zipf parameter θ	1.1 - 2.0	1.5
Multicast Pull Threshold	2	
System Bandwidth	100000 bytes/sec	100000 bytes/sec
Request rate λ	250 / sec	250/sec
Re-configuration period	1 - 60 sec	1 sec
Total items n	100 - 10000	1000
Total Requests Made	10000	
Hot Spot Move Type	Off, Small, Big	Off
α	1.1 - 4.2	2

Table 2: Simulation Parameters

The other parameters used throughout our experiment are found in Table 2. The bandwidth split refers to the amount of the bandwidth (as a percent) given to pull and push, respectively. The zipf parameter is the amount of skewness in the popularity of documents, with 1.1 being more uniform and 2.0 being more skewed. The request rate is the average number of requests made each second into the server, to create an overload situation. The re-configuration period is the amount of time between execution of the document selection module in the server. The other parameters refer to experiment specific items and are addressed within the experiments themselves. Unless otherwise stated, all the parameters used in the various experiments are the default values shown in the table.

In each experiment, we measured the average latency at the client over a set of requests. Note that in measuring latency, there are two factors that must be taken into account, delay at the server and network delay. Because our protocol focuses on actions at the server, the main delay that is measured in our average latency is the server side delay. By putting the server and client on the same system, we effectively eliminated the network delay by creating a controlled environment, and hence network delays are not really taken into account in our results. Different networks may effect our results in various ways, but our savings at the server itself will still be as reported in the experiments below.

3.1 Experiment 1: Validation of our Document Classification and Bandwidth Division Algorithm

Figure 2 shows the effects of various values of α on the average latency of Algorithm 2. The curve in Figure 2 is jagged because an infinitesimal change in α can have a discrete effect in the number of items pushed. Figure 2 shows that the value of α that minimizes average latency is between 2.0 to 3.0. We adopt $\alpha = 2.0$ in the rest of the paper — although this is not the actual minimum, any value in the range produces similarly good results. Note that as α changes in figure 2 our system adjusts the bandwidth division and document classification to maintain optimality. This in part explains why the average latency is near optimal for a relatively wide range of α.

Figure 3 can be interpreted as a brute force search for a good bandwidth split and document classification by trying several closely spaced values of k and pushBW. In the chart legend, the first number in the bandwidth split refers to pull. In addition to the points plotted in the figure, we verified that if less than half of the bandwidth was devoted to pull, the latency was suboptimal. In this scenario, Algorithm 2 assigns the most popular 7 documents on the push channel, and allocates 63% percent of the bandwidth to push. The figure shows the algorithm's outcome with a circular point and an arrow pointing to it. The solution produced by our algorithm is better than any other point in the diagram. More specifically, our algorithm chose a split of 63/37 and the closest brute force curve in the figure is the 65/35 curve. The 65/35 line was also the lowest in the graph. Algorithm 2 chose $k = 7$ point as the number of push documents, which is also the minimum point on the 65/35 curve. Thus, Algorithm 2 chose a better bandwidth split than the brute force approach and a document classification that was just as good.

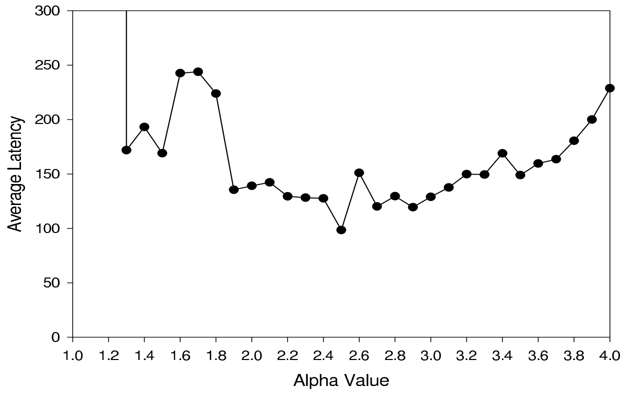

Figure 2: Effects of various α values on average latency

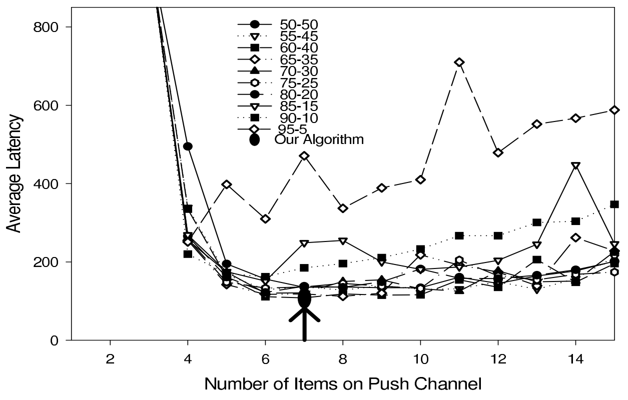

Figure 3: Demonstrating the optimality of Algorithm 2 for document classification and bandwidth division. The arrow points to the single point found by the algorithm.

Let $G(k)$ be the average latency if the k most popular documents are placed on the push channel. The function $G(k)$ is a weighted average of the average latency for pushed documents and the average latency for pulled documents. A graph showing an idealized $G(k)$ from [29] is shown in Figure 4. The function $G(k)$ has a unique local minimum, which can be be found by local search [29]. Figure 4 shows that the minimum of $G(k)$ is to the right of the intersection of the push and pull curves. In this case, pulled documents would have lower latency than pushed documents. The actual curve that we obtained from our experiments is shown in Figure 5. Notice that the minimum of $G(k)$ is to the left of the intersection of the push and pull curves, and thus pushed documents have lower latencies than pulled documents. Further, the minimum of $G(k)$ occurs at a relatively small value of k, and thus complicated hierarchical schemes for the push channel may not be useful in this setting. The location of the minimum is due to two complementary reasons. First, the most popular items are chosen for push and are also those to which a Zipf (or Zipf-like) distribution gives substantially more weight. Therefore, if a solution favors multicast push, it will also have the largest impact on the globally average delays. Second, the unicast pull curve levels off and, from that point on, the exact choice of k has little impact on pull delay. In other words, pull delays are practically minimized at the point $\bar{k}$ where the pull curve flattens out. However, $\bar{k}$ precedes the intersection of the pull curve with the push curve, and so the overall minimum occurs before that intersection.

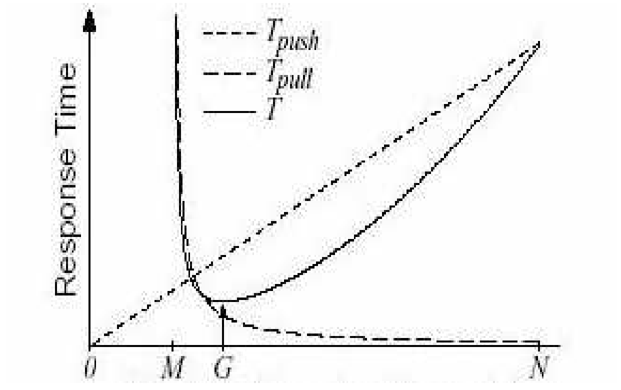

Figure 4: Relation of Push and Pull latencies as number of items pushed is changed, according to Stathatos *et al.*

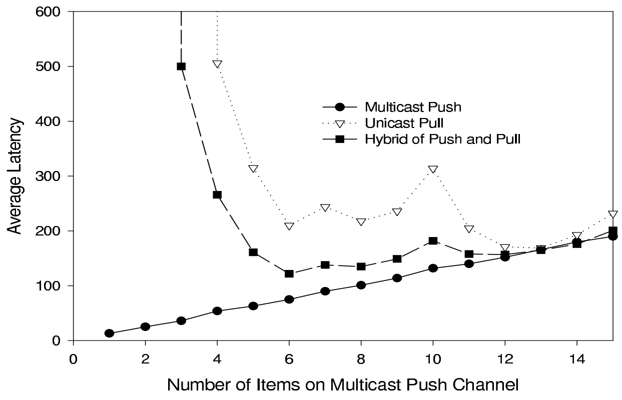

Figure 5: Relation of Push and Pull latencies as number of items pushed changes according to our experiments

In conclusion, Algorithm 2 was shown to be better than the best value returned by a brute force search. Furthermore, the integrated algorithm led to a behavior of the push and pull curves that differs qualitatively and quantitatively from previously published work, e.g., in terms of the relative behavior of push and pull delays.

3.2 Experiment 2: Report Probabilities

In order to determine the usefulness of our proposed push popularity scheme, we compare it to a solution found in a comparable work to our own. The solution for the push popularity problem proposed in [29] was to occasionally drop each pushed document i off of the push channel so that clients would have to make explicit requests to i. However, there is a danger that these explicit requests for i could overload the server. Thus, in [29] it was recommended that i should be dropped as short of a period of time as possible. The shortest possible time the the document can be dropped is one broadcast cycle. However, we show here that even such a short drop disrupts the server, while our proposed method does not suffer from such disruptions.

Figure 6 shows the average latencies around the broadcast cycle T when the most popular item is dropped from the push channel. The figure shows a performance degradation for about 5 broadcast cycles. Basically, looking at the graph shows that before the drop occurs, the system is in a steady state of response times. However, once the item is dropped down the clients are no longer getting requests off the push channel. Instead, they must make requests directly to the server. Based on the Zipf distribution, as mentioned earlier, the bulk of requests were for items that were on the push channel. Therefore, dropping an item down causes a brief but substantial influx of requests to the server. This brief surge causes response times for requests during the given broadcast cycle and a few subsequent cycles to suffer while the server recovers and returns to its steady state.

Figure 7 shows the average latency over the next 5 broadcast cycles when the i^{th} most popular document is dropped from the push channel for one broadcast cycle. The flat line represents the average response time using our method for push popularity. If the most popular document is dropped, then we see a 35% increase in average latency over the next 5 broadcast cycles. If the 6th most popular document is dropped, we see an 8% increase in average latency over the next 5 broadcast cycles. This increase is in comparison to using the simple yet effective scheme we proposed of simply including a popularity estimator with the broadcast index.

In fairness, our proposed method has the disadvantages that it requires extra space in the broadcast index and it slightly increases the request rate at the server during all broadcast cycles.

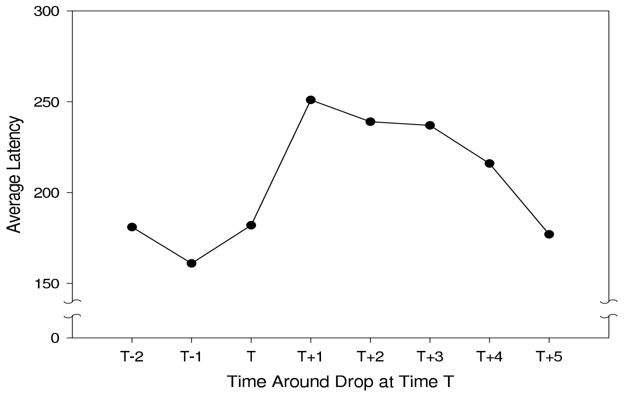

Figure 6: Effect on latency of demoting an item.

3.3 Experiment 3: To Multicast Pull or Not to Multicast Pull

We now compare the performance of our system with multicast pull turned off versus multicast pull turned on. We assume a static distribution, that is, document popularities do not change over time. The results of this experiment are shown in Figure 8.

Figure 8 shows the average latencies with multicast pull on, and

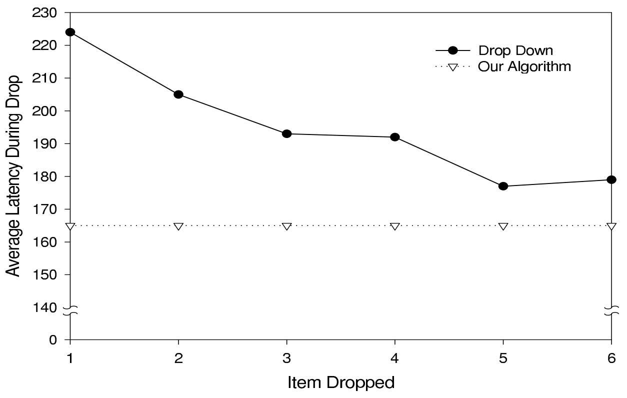

Figure 7: Drop down method versus our probability method.

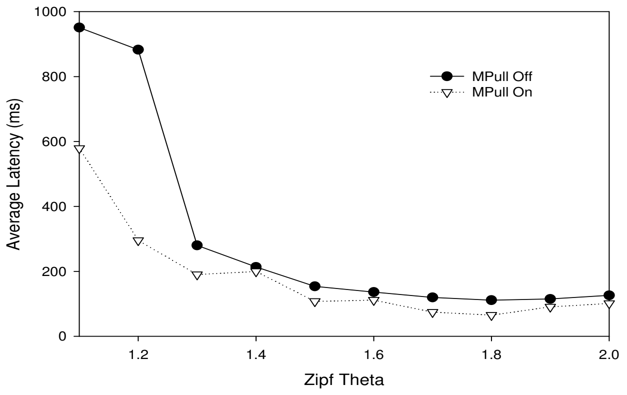

Figure 8: Average latencies for multicast pull On vs. multicast pull off for static access patterns

with multicast pull off, for various θ values. We found a reduction in average latency of 66% for $\theta = 1.2$, and a reduction of 30% for $\theta = 1.5$. For example, when the $\theta = 1.5$, the average latency decreases from 153.9ms with multicast pull off, to 107.6ms with multicast pull on. For the higher θ's, the difference in average latencies was not statistically significant. These results conform to intuition. The distributions for smaller θ's are more heavily tailed. Thus for a smaller θ you would expect more requests to arrive for pulled documents, making it more likely you would get near simultaneous requests for the same documents, and thus you would expect multicast pull to be of greater advantage.

Another advantage of multicast pull is that it decreases the standard deviation of the observed latencies. Thus, with multicast pull turned on, fewer requests will have to wait for extreme lengths. The standard deviations are shown in Figure 9. For $\theta = 1.5$, the average latency with multicast pull off was 153.9 milliseconds, with standard deviation of 308 ms. In contrast, with multicast on, not only was the average latency significantly better at 107.6 milliseconds, but the standard deviation also improved to 226 ms.

3.4 Experiment 4: Multicast Pull with Moving Hot Spot

We examine the advantage of using multicast pull when the popularities of documents changes over time, but the Zipf parameter θ stays fixed. We look at two modes of change:

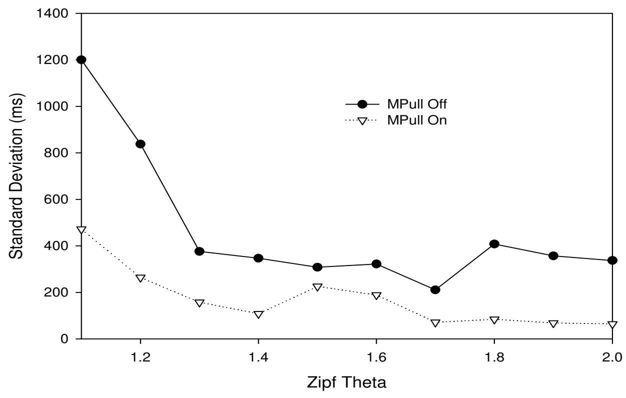

Figure 9: Standard deviations in latencies for multicast pull on vs. multicast pull off for static access patterns

- The first mode is when the popularities change gradually over time (*small move*). The first mode would reflect a gradual client shift in interest over time.
- The second mode is when there is a sudden phase change in the location of the hot spot (*big move*). The second mode would reflect a sudden change in client interest, perhaps in response to an important event.

Document classification is a relatively expensive operation, requiring time linear in the number of documents, and the server can not afford to always be invoking the document classification algorithm. We show in this subsection that until the document classification is invoked, multicast pull helps provide an intermediate form of scalability.

Figures 10 and 11 show the resulting average latencies resulting from gradual changes in popularity while varying one of θ or α and keeping the other parameter fixed to its default value. In these experiments, periodically each document would swap popularities with the next most popular document with probability 1/2. For example, with probability 1/2, the second most popular document would become the third most popular document. For these experiments the access probabilities change every 500 requests received from the monitored client. These 500 requests do not include requests made by the request filler.

The most interesting feature in Figure 10 is that multicast pull is more helpful as θ increases. More precisely, the relative improvement one achieves in average latency when using multicast pull increases as θ increases. For example, when $\theta = 1.5$, multicast pull shows an improvement in average latency of 44.6%(from 217.9 ms to 120.9 ms), at $\theta = 1.7$ the improvement in average latency is 47.1%(from 228.9 ms to 121.1 ms), and at $\theta = 2.0$ the improvement in average latency is 50.3%(from 243.5 ms to 121.2 ms). The explanation is that for large θ's, most of the probability is in the most popular items, so if a pull document should become more popular, it will receive many requests before the server next invokes the document classification algorithm.

The most interesting feature of Figure 11 is that the best choice of α for small moves is larger than it is for a static distribution. Recall that the optimal choice of α for a static distribution was in the range 2.0 to 2.5. In this experiment the optimal choice for α

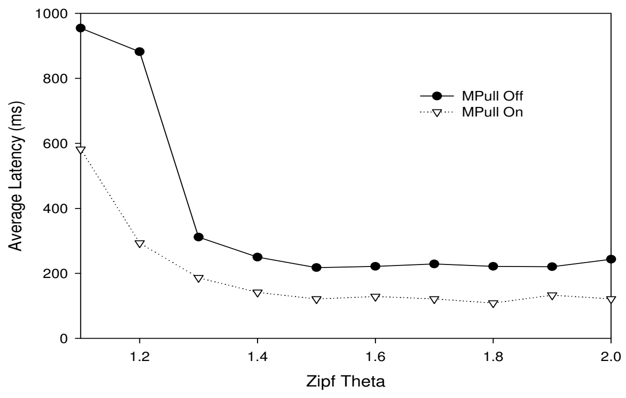

Figure 10: Average latency for multicast pull On vs. multicast pull off for small move access patterns

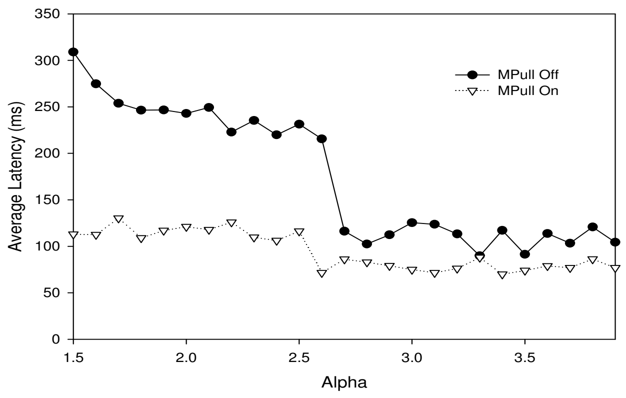

Figure 11: Average latency multicast pull on vs. multicast pull off various α's and small move access patterns

is in the range from 3.0 to 4.0. In this case, setting $\alpha = 2$ has an average latency of 121.2ms while with $= 3.5$ the average latency is 74ms, a decrease of 39%. The explanation is that as there is a shift in popularity, the popularity of the pulled documents will be greater than estimated, and thus obviously, pull should be even more over provisioned.

Figure 12 shows that once again multicast pull reduces the standard deviation of the observed latencies. For $\theta = 1.5$, the standard deviation of the latencies decreases 60% from 235ms to 93ms. Note that the reduction in the standard deviation is greater than in the case of static access probabilities. The reason for this is because multicast pull provides some scalability when the pulled documents become popular.

Figures 13 and 14 show the results of a similar experiment to above but in this case we are looking at big moves in the popularity of an item. In a Zipf distribution, each p_i is proportional to

$$\frac{1}{(1 + (i - b) \bmod n)^{\theta}}$$

for some base b, which we normally think of as 0. Periodically we change the value of b. This simulates a sudden shift if the clients' document interests. In this case, we again see that using multicast pull is a significant win for larger θ. As one would expect, for both methods, the latencies are higher than in the slowly moving hot spot experiment. Using multicast pull we see a reduction in

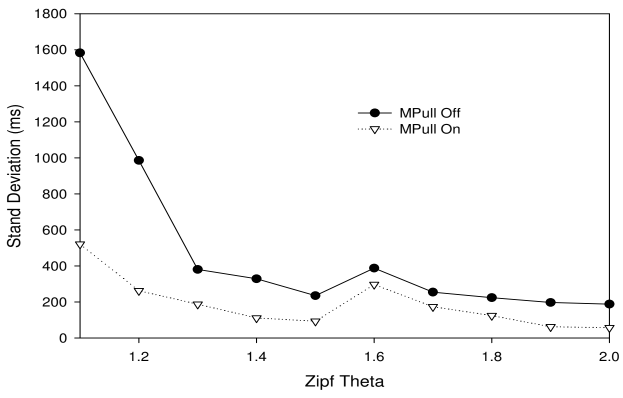

Figure 12: Standard deviation on latency for multicast pull on vs. multicast pull off for small move access patterns

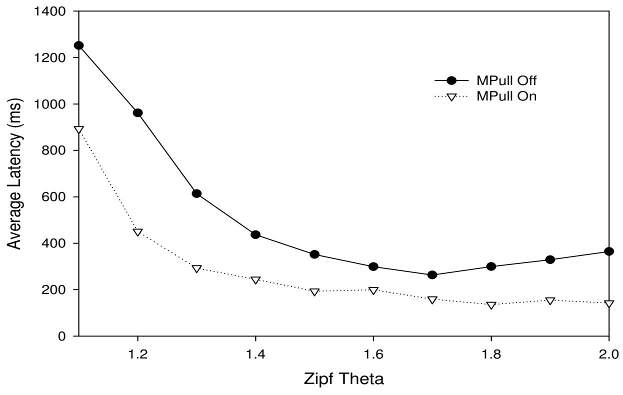

Figure 13: Standard deviations for multicast pull on vs. multicast pull off for big move access patterns

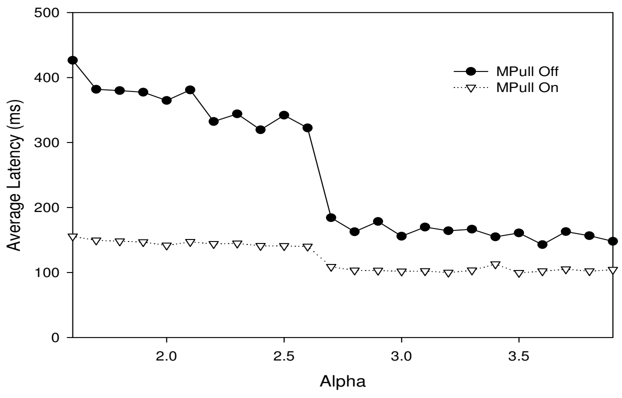

Figure 14: Average latencies for multicast pull on vs. multicast pull off various α's and big move access patterns

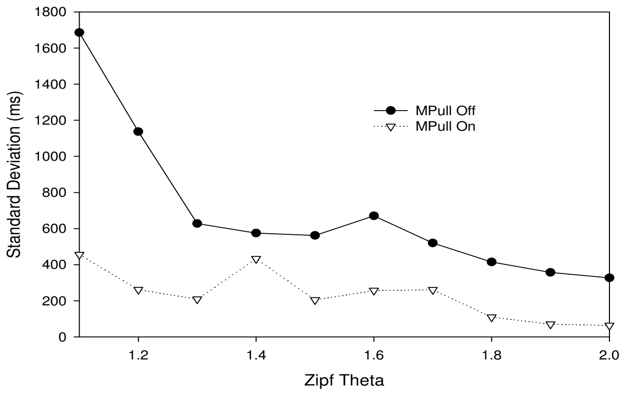

Figure 15: Standard deviations for multicast pull on vs. multicast pull off for big move access patterns

average latencies. For $\theta = 1.5$ the reduction is 45% from 351.8 ms to 193.2 ms, for $\theta = 1.7$ the reduction is 40% from 263.4 ms to 159 ms, and for $\theta = 2$ the reduction is 61% from 364.7 ms to 141.8 ms. We also see again that the over provisioning factor should be greater than for static documents.

Figure 15 shows that for big moves, multicast pull reduced the standard deviation of the latencies even more dramatically for small moves. For $\theta = 1.5$ the standard deviation has decreased 63% from 562 ms to 205 ms, and for $\theta = 2$ the decrease was 81% from 327 ms to 63 ms. The reason that multicast pull reduces the standard deviation more for big moves than for small moves is that the scalability that multicast pull provides becomes more important as the pulled documents become more popular.

Average latency is by far the most commonly used quality of service (QoS) metric in the literature. The metric is simple and intuitively appealing. However, it is also well known that average latency is generally not the ideal system metric in that the solution that optimizes average latency may starve some jobs. Allowing jobs to starve is considered bad system behavior. Ideally one would want a metric that balances the competing demands of optimizing for the average and avoiding starvation. The standard solution is to use the ℓ_p norm for small p. The ℓ_p norm is $\left(\sum_{i=1}^{n} \frac{x_i^p}{n}\right)^{1/p}$. For example, the standard way to fit a line to collection of points is to pick the line with minimum least squares, equivalently ℓ_2, distance to the points, and Knuth's TEXtypesetting system uses the ℓ_3 metric to determine line breaks. The ℓ_p, $1 < p < \infty$, metric still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the ℓ_p norm more severely penalizes outliers than the standard ℓ_1 norm.

Figures 16 and 17 show what the effect of varying α has on the difference between having multicast pull on and off for the ℓ_2 norms latencies. We set $\theta = 2.0$. For $\alpha = 2$ and small moves, the ℓ_2 norm of the latencies decreases by 78% from 475.2 ms with multicast pull off to 138.7 ms with multicast pull on. At $\alpha = 3.5$ and small moves, the decrease is 42% from 277.1 ms with multicast pull off to 160.6 ms with multicast pull on. We found similar results for big moves. These results are shown in Figure 17. For $\alpha = 2$, we see a reduction in the ℓ_2 of latencies of 71.5% from 1025.3 ms to 292.8 ms. For $\alpha = 3.5$ we see a reduction of 78% from 591.1 ms to 128.64 ms. Once again this shows the scalability provided by multicast pull. When the server become highly loaded during popularity shifts, the system without multicast pull, experiences increased latencies until the document selection thread is invoked.

3.5 Experiment 5 - Multicast Pull Advantage with varying Time between Reconfiguration

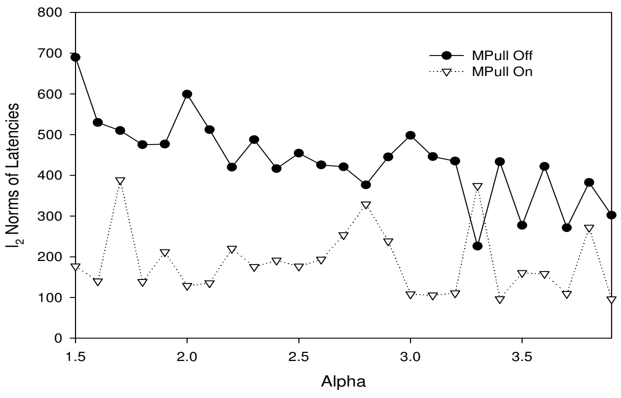

Figure 16: ℓ_2 norms of latencies for various α and small move access patterns

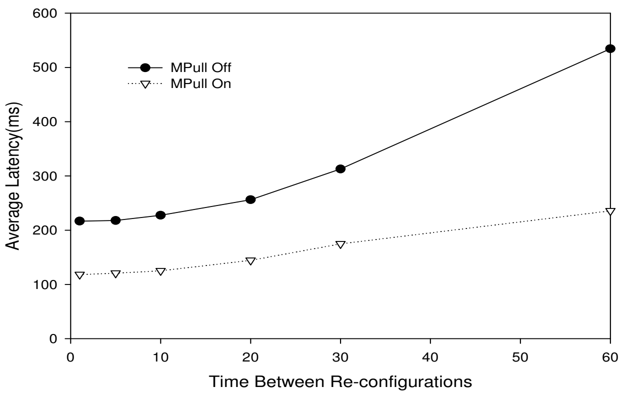

Figure 18: Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for small move access patterns

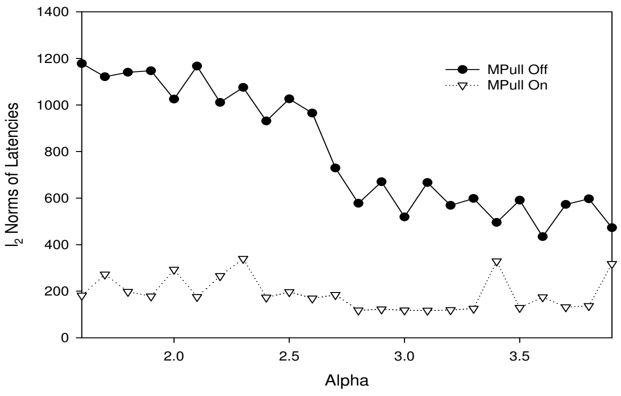

Figure 17: ℓ_2 norms of latencies for various α's and big move access patterns

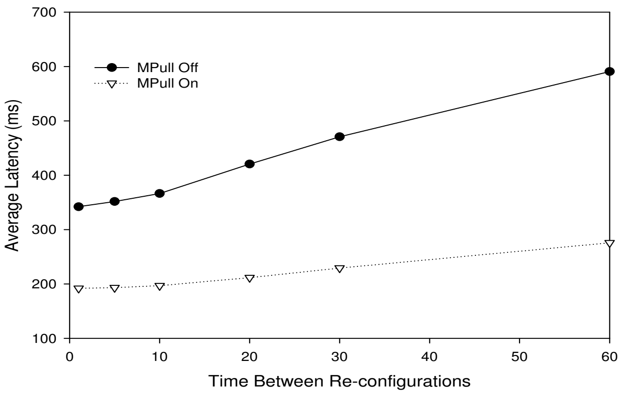

Figure 19: Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for big move access patterns

In this experiment, we examine the advantage of using multicast pull when the time between invocations of document selection changes. For this experiment, we set the θ to 1.5.

Figures 18 and 19 show the results of the experiment. As one would expect, using multicast pull is more advantageous when reconfigurations are less frequent. The obvious reason is that it is taking the system longer to adjust to the changes in user preferences, and therefore there are many requests coming in that have to be handled through pull.

Using multicast pull we observe a reduction in latency for small moves of 46% when the reconfiguration is every 10 seconds, 45% when the reconfiguration is every 20 seconds, and 55.6% when the reconfiguration is every 60 seconds.

For big moves we observe a reduction in latency of 46.4% when the reconfiguration is every 10 seconds, 50% when the reconfiguration is every 20 seconds, and 53.4% when the reconfiguration is every 60 seconds.

Notice the trade off that exists between waiting too long to run the reconfiguration and the average latencies. The longer that is waited to reconfigure the system, the worse the response times for clients get. However, using multicast pull can help maintain lower latencies when the system is slow in adapting to changes in the request patterns.

4. RELATED WORK

The general idea behind a Multicast Pull channel is not new, but the way we are using it in this paper is different than in work previous done on the topic. One of the key ideas behind Multicast Pull is that it is basically a channel that pushes data that was previously requested by users. In [16] the idea of multicast pull is presented in terms of 1-to-N multicasting, where the data is sent to a specific set of clients that expressed interest in the data. Other work [1, 18, 4, 31] also looks into the idea of using user requests to determine what should be multicasted out. This set of scheduling schemes and multicast pull papers differ in both approach and purpose to this paper. These papers are looking out how to schedule items on a single multicast (broadcast) channel using the requests from users, a topic we are not addressing. We are looking at the usefulness of having a Multicast Pull channel available in addition to a Multicast Push (broadcast) channel for popular documents and a Unicast channel to handle client requests in normal web server fashion.

Work that is closer in nature to what we are doing are the DBIS-toolkit[5] and Air-Cache[28]. The DBIS project is similar to our middleware in which we use multiple data dissemination channels. DBIS uses multicast and unicast to address client needs however looks to do so by translating between the different dissemination methods into one overlay network. Additionally, the format of our system, in terms of how the channels are used, is focused more on performance and robustness than the DBIS system. Air-Cache also uses several channels, namely Multicast Push and Unicast, to service client requests. This is done in a similar fashion to our system, however again no Multicast Pull channel exists.

The document classification problem was introduced in [29]. In addition to directly related work, some other work has been done addressing the issue of hot and cold documents and of bandwidth division, though not in the context we are describing. In [1, 18, 4, 31] the issue of mixing pull and push documents together on a single broadcast channel is examined. The idea is that popular documents are similarly considered hot, and are continuously broadcast while all other documents are cold. These documents are request through a back channel and scheduled for broadcast. Similarly, in [1] the authors discuss how to divide the broadcast channel bandwidth between hot and cold documents. The main difference between previous work and ours is previous work deals with a broadcast environment with a single channel and focuses on scheduling items, not how to divide them into hot and cold. We are looking into the division of both documents and bandwidth to minimize latency.

The hybrid scheme relies on estimates of the popularity of documents in the web site because popularity determines the assignment of documents to dissemination modes. Popularity estimation can be approached separately for pulled and for pushed documents. Pull popularity can be solved in sub-linear space by monitoring the client request stream [12]. As for push popularity, the problem is complicated by the absence of a client request stream. One solution is to occasionally drop each pushed document from the push channel, thus forcing clients to send explicit requests. Such requests can then be counted and the document popularity estimated [29]. A related problem is multicast group estimation [23], which can be specialized as follows in our context: remove a document from the multicast push channel and re-insert it as soon as the first request for that document is received. The document popularity can be estimated by the length of time it takes for the first client request to reach the server.

5. CONCLUSION

Scalability has been the number one problem in the efficient dissemination of data in the Internet. Multicast Push and peer-to-peer techniques have been shown that offer the best promise to achieve scalable data dissemination. Our overall research goal is to combine the advantages of these two approaches in a peer-to-peer system with built-in multicast data dissemination. In this paper, we argued for the use of multicast pull in conjunction to multicast push and described how this has been achieved in our prototype system.

More specifically, we proposed (1) a simple algorithm for the push popularity problem that is more scalable in estimating the popularity of pushed documents and (2) the essentially optimal algorithm for document classification and bandwidth division. We validated our proposed algorithm experimentally.

The last major contribution of our paper is the quantification of the advantage of including a multicast pull component. We showed that multicast pull is of modest, but measurably advantage at the level of 5%-25% reduction in average latency compared to pure multicast push, when the popularity distribution is static. But multicast pull is of significant advantage when the popularity distribution is dynamic. In such a dynamic environment, multicast pull provides an intermediate form of scalability until the expensive document classification algorithm is invoked. These results are also applicable in a mobile and wireless environment in which the primary mode of communication supports broadcasting. In fact, we believe that these results contribute toward realizing a wireless web in which users are mobile.

We are currently porting our system over a multicast capable peer-to-peer network, such as Scribe [10]. In such a system our middleware nodes will become peers acting as reverse proxies to multiple web/database servers. Hence our middleware nodes will assist each other to deal both with the dissemination of data as well as dealing with failures.

6. REFERENCES

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull data broadcast. In *ACM SIGMOD*, 1997.

[2] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *ACM/IEEE MobiCom*, pages 43–54, 1998.

[3] D. Aksoy and M. Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Transactions on Networking*, 7(6):846–860, 1999.

[4] D. Aksoy and M. Franklin. Rxw: A scheduling approach for large-scale on-demand data broadcast. *ACM/IEEE Transactions on Networking*, 7(6):846–860, 1999.

[5] M. Altinel, D. Aksoy, T. Baby, M. Franklin, W. Shapiro, and S. Zdonik. Dbis toolkit: Adaptable middleware for large scale data delivery. In *ACM SIGMOD*, 1999.

[6] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, and L. Brown. Dbcache: Database caching for web application servers. In *ACM SIGMOD*, 2002.

[7] Y. Azar, M. Feder, E. Lubetzky, D. Rajwan, and N. Shulman. The multicast bandwidth advantage in serving a web site. In *3rd NGC*, pages 88–99, 2001.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Infocom*, 1999.

[9] R. Càceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance/*, 1998.

[10] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[12] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking frequent items dynamically. In *Proceedings of Principles of Database Systems*, 2003.

[13] A. Datta, K. Dutta, K. Ramamritham, H. M. Thomas, and D. E. Vandermeer. Dynamic content acceleration: A caching solution to enable scalable dynamic web page generation. In *ACM SIGMOD*, 2001.

[14] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *ACM SIGMOD*, pages 97–108, 2001.

[15] H. D. Dykeman, M. Ammar, and J. W. Wong. Scheduling algorithms for videotex systems under broadcast delivery. In *International Conference on Communications*, pages 1847–1851, 1986.

[16] M. Franklin and S. Zdonik. " data in your face ": Push technology in perspective. In *ACM SIGMOD*, 1998.

[17] Y. Guo, M. Pinotti, and S. Das. A new hybrid scheduling algorithm for asymmetric communication systems. *ACM SIGMobile Computing and Communications Review*, 5(3), 2001.

[18] A. Hall and H. Taubig. Comparing push- and pull-based broadcasting or: Would "microsoft watches" profit from a transmitter? *Lecture Notes in Computer Science*, 2647, 2003.

[19] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *OSDI*, pages 197–212, 2000.

[20] A. Labrinidis and N. Roussopoulos. Webview materialization. In *ACM SIGMOD*, 2000.

[21] A. Labrinidis and N. Roussopoulos. Balancing performance and data freshness in web database servers. In *VLDB*, 2003.

[22] Q. Luo, S. Krishnamurthy, C. Mohan, H. Woo, H. Pirahesh, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *ACM SIGMOD*, 2002.

[23] J. Nonnenmacher and E. W. Biersack. Scalable feedback for large groups. *IEEE/ACM Trans. Netw.*, 7(3):375–386, 1999.

[24] V. Padmanabhan and L. Qiu. The context and access dynamics of a busy web site: Findings and implications. In *ACM SIGCOMM*, 2000.

[25] P. Rosenzweig, M. Kadansky, and S. Hanna. The java reliable multicast service: A reliable multicast library. SMLI TR-98-68, Sun Microsystems, 1998.

[26] M. A. Sharaf and P. K. Chrysanthis. Facilitating mobile decision making. In *Proc. of the ACM Workshop on Mobile Commerce*, pages 45–53, 2002.

[27] M. A. Sharaf and P. K. Chrysanthis. Semantic-based delivery of olap summary tables in wireless environments. In *CIKM*, pages 84–92, 2002.

[28] K. Stathatos, N. Roussopoulos, and J. Baras. Adaptive data broadcasting using air-cache. In *First International Workshop on Satellite-based Information Services (WOSBIS)*, 1996.

[29] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. In *VLDB*, pages 326–335, 1997.

[30] P. Triantafillou, R. Harpantidou, and M. Paterakis. High performance data broadcasting: A comprehensive systems' perspective. In *MDM*, pages 79–90, 2002.

[31] P. Triantafillou, R. Harpantidou, and M. Paterakis. High performance data broadcasting systems. *Mobile Networks and Applications*, 7:279–290, 2002.