World Wide Web (2008) 11:313–337
DOI 10.1007/s11280-007-0039-4

Improving the Hybrid Data Dissemination Model of Web Documents

Jonathan Beaver · Kirk Pruhs · Panos K. Chrysanthis · Vincenzo Liberatore

Received: 23 November 2004 / Revised: 15 June 2007 /
Accepted: 20 November 2007 / Published online: 13 June 2008

Abstract One of the major problems in the Internet today is the scalable delivery of data. With more and more people joining the Internet community, web servers and services are being forced to deal with workloads beyond their original data dissemination design capacity. One solution that has arisen to address scalability is to use multicasting, or push-based data dissemination, to send out data to many clients at once. More recently, the idea of using multicasting as part of a hybrid system with unicasting has shown positive results in increasing server scalability. In this paper we focus on solving problems associated with the hybrid dissemination model. In particular, we address the issues of *document popularity* and *document division* while arguing for the use of a third channel, called the *multicast pull channel*, in the hybrid system model. This channel improves performance in terms of response time while improving the robustness of the hybrid system. We show through extensive simulation using our working hybrid server the usefulness of this additional channel and its improving effects in creating a more scalable and more efficient web server.

Keywords hybrid system design · data dissemination · scalability · multicasting

J. Beaver · K. Pruhs · P. K. Chrysanthis (✉)
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
e-mail: panos@cs.pitt.edu

J. Beaver
e-mail: beaver@cs.pitt.edu

K. Pruhs
e-mail: kirk@cs.pitt.edu

V. Liberatore
Division of Computer Science, Case Western Reserve University,
10900 Euclid Ave., Cleveland, OH 44106-7071, USA
e-mail: vincenzo.liberatore@case.edu

1 Introduction

With more and more people joining the Internet community, web servers and services are being forced to deal with workloads beyond their original data dissemination design capacity. This has led to the recognition that scalable delivery of data is one of the largest challenges in the Internet today. One solution is to use a collection of servers as in a server farm [9]. Another solution is a content delivery network such as Akamai, which replicates the data at various servers across the Internet [24]. However these solutions are heavy-weight, and reasonably costly for the server owner. Furthermore, these solutions require significant overhead and upfront planning in order to set them up.

There are many instances where the content provider needs scalability but can not justify the effort in planning, the overhead of setting up, and the cost of using these heavy weight solutions. This is often a result of flash crowds at a normally modestly popular site. One example is the non-commercial site factcheck.com, which was inundated after Dick Cheney mistakenly mentioned the site, when he meant to mention factcheck.org, during the 2004 US vice presidential debate [12]. The owners of `factcheck.com` were unable to handle the load, and would not have wanted to pay the expense if they could. They ended up redirecting traffic to a site run by billionaire George Soros. Another example is the site xprize.org, when the Xprize for private space travel was won on October 4 [29]. Clicking on the Google logo at google.com redirected the browser to `xprize.org` bringing down the normally lightly loaded non-commercial site. In both cases, the web site operators needed a cheap, quick, and easy scalability solution.

One possible solution is a hybrid multicast based system [2, 4, 5]. Multicast can be implemented with overlay networks [20], peer-to-peer networks [11, 32] (Napster, http://www.napster.com), or with IP multicast. In *multicast push* documents are multicast cyclically and repeatedly from a server to clients in the absence of client requests [18, 31, 33, 34]. A multicast push document is delivered to all hosts subscribing to the multicast channel and therefore its delivery consumes bandwidth and computation at every participating site. Additionally, if multicast push were to be used extensively, the multicast channel could be clogged with unpopular items, leading to increased overall response times for clients. Therefore, a more prudent approach is to have multicast push be reserved for popular documents, and less popular data can be disseminated with the traditional request-response *unicast pull* approach [2, 14, 31]. This we refer to as a *hybrid system* approach to data dissemination because of its combination of multiple delivery types to handle the distribution of data.

The hybrid system approach is able to increase scalability while keeping low response times at clients, but it has several issues that must be resolved to use it. One very important issue is the handling of mispredictions of what items belong on which distribution channel occurs or when the popularity of items shifts. When these events occur, the system will begin to receive many requests over the unicast channel, requests that should have been handled by the multicast push channel. This can lead to increasing response times and server slow down until the right items are placed again on the appropriate channel.

In this paper, we address this issue of misprediction and popularity shifts by adding into the hybrid data dissemination model an additional third channel which we refer to as the *multicast pull* channel. This channel combines the feature of multicast with the use of explicit client requests. In multicast pull, clients request documents from a server, which in turn multicasts its reply to all connected clients. In other words, the server sends a reply only once to all clients that made the same request. As a result, multicast pull is more scalable than the traditional unicast pull because it can implicitly aggregate multiple requests into one reply. However, multicast pull is less scalable than multicast push, which avoids any per-request processing. Multicast pull should be devoted to documents whose popularity is intermediate between that of the multicast push and of the unicast pull data sets. The adoption of multicast pull poses the fundamental difficulty of whether such intermediate documents can be usefully individuated or even if they exist in the first place.

We present in this paper a detailed evaluation of the implications of multicast pull. Our primary objective is to quantify the improvements afforded by multicast pull in terms of robustness, scalability, and performance. The evaluation will include extensive experimentation on a full-fledged data dissemination middleware. This middleware is also available for downloading and testing with any web server, following the terms of open source distribution, at http://db.cs.pitt.edu/software/mbdd/. Our evaluation employs a prototype middleware that supports a three-tier multicast dissemination scheme and acts as a reverse proxy to a Web server for the delivery of documents. We will show that using this additional channel both decreases the latency perceived by clients and increases the robustness of the hybrid system model, especially during times of popularity shifts and mispredictions.

The remainder of the paper is organized as follows. In Section 2 we will provide background on data dissemination including related work and motivate the use of hybrid systems. Section 3 describes our solutions to the problems associated with hybrid systems and includes a description of our working hybrid middleware. We then present our experimental methodology in Section 5 followed by experiments and results in Section 6. Finally, we conclude the work in Section 7.

2 Data dissemination

Data dissemination is the process of getting requested data out to clients in a timely manner. The way in which the actual dissemination is done is dependent upon the system model being used. In this section we will examine three models that exist for data dissemination, the pull-based unicast model found in many web servers, the push-based multicast model and the hybrid model which includes both unicast and multicast. A quick summary of the different dissemination models is found in Table 1.

Table 1 Comparison of data dissemination models.

Dissemination mode	Latency	Scalability
Pull-based unicast	Low	Low
Push-based dissemination	High	High
Hybrid multicast pull	Medium	Medium
Hybrid multiple channels	Low	High

2.1 Pull-based data dissemination model

In the pull-based data dissemination model, clients make requests directly to the server and then await responses over the same channel. This is known as a one-to-one data dissemination model, because there is one client making the request and logically one server responding to that request (although the server may be part of a cluster, there is still only one server of that cluster servicing the request). This model tends to perform well when the server load is low, and produces very low latency on the side of the client.

The issue that exists with this model, however, is scalability. The reason scalability is a problem is that as more and more clients make requests, the server is forced to become a bottleneck, as it is no longer able to service requests at the rate at which requests are coming into the server. This in effect leads to response times that are ever increasing towards infinity [23]. In web servers, it may be the case that instead of having response times scale to infinity, requests are just dropped when the rate of incoming requests gets too high or in some extreme cases, the web server itself crashes and all requests are dropped.

2.2 Push-based data dissemination model

In the push-based data dissemination model, which we refer to as being done through multicasting, clients are no longer required to make requests to the server for data. Instead, the data is pushed out by the server and delivered to all clients interested in the data that have joined the multicast tree. This delivery model is what is known as one-to-many because there is one send out being done by the server but it is reaching many clients. This model tends to perform very well in situations where high scalability is needed because requests are not made directly into the server and thus will not overload it.

The issue that exists with a push-based model is that while it scales, it also must service all requests that exist from all clients, since no clients are making requests to the server. This can lead to clients waiting a long time to receive the data they want, since much of the data sent over the channel will not be relevant to a given client. In addition, using a push-based model means that all items in the server must be properly scheduled [17, 21, 22] to both service all requests and keep response times at a minimum. This issue is even more complicated when popularities of items begin to shift as push-based data dissemination has no mechanism for feedback from clients, which could again adversely effect the latency perceived by clients.

2.3 Hybrid data dissemination model

In the hybrid data dissemination model, multiple channels are instituted to improve both performance and scalability over the previously defined models. In particular, there are two hybrid models we will be examining, the first being what could be viewed as the pure multicast pull model and the second being a hybrid system which includes multiple channels over which requests and responses are sent.

2.3.1 Multicast pull

The hybrid system which we refer to as multicast pull can be seen as a normal multicast push system but with an additional unicast based back-channel over which requests can be sent. This model has become very popular especially in the wireless environment, where broadcast is used to distribute data to all clients. The main idea is that unlike pure push based approaches, this system provides a way for clients to make requests to the server. The server can then use these requests to schedule which items to place on the push channel at what times. Scheduling in these systems is not a trivial problem, and has been the subject of much prior research (for example, [1, 3, 4, 16]).

Some of these approaches in multicast pull type hybrid models use a set of popular, or hot, documents which are always multicasted out to clients with no explicit requests made by the clients. The remaining documents are only multicasted out after receiving requests from clients, at which point they are scheduled to be sent out intermixed with the hot documents. There are several issues that arise in such a system, with the important one to note here is that clients still have to wait for the requests of other clients to appear on the multicast channel before their own appear. Just as important, because there is a set of popular data being continuously sent out, clients for cold documents will have very long response times in many cases because they may have to wait for popular data to be multicasted multiple times before their own data is sent out.

2.3.2 Multiple channel hybrid model

The multiple channel hybrid model disseminates data through the use of multiple channels, with data actually being distributed out over every channel. In particular, two channels exist for data distribution, a multicast push channel and unicast pull channel. On the multicast push channel hot documents are placed and pushed out to all clients in cycles. These documents do not require explicit requests from clients and are designed to increase scalability by not requiring an influx of requests to appear for the most popular documents on the server. The unicast pull channel is used to respond to requests like a normal unicast server would. The requests made over this channel are for the cold documents in the system. The clients are replied to directly, in contrast to the multicast pull model where the requests would be scheduled to be sent out over multicast. This leads to lower response times for unicast clients because they are responded to right away while push clients only have to wait for a small set of popular data to be sent out and not have cold documents intermingled in the hot channel.

While this model provides nice scalability and low response times, it still has several issues which need to be addressed. The major issues are how to decide what items to place on the hot channel and which to place on the cold channel, how to determine the popularity of items on the push channel, and what to do in cases where the wrong set of items is on the push channel, meaning the popularity of items has been mispredicted or the popularity of items has shifted over time. These are important issues because they can lead to poor response times at clients and possible server overload when the misprediction is occurring on all documents. Therefore, these issues should be addressed in any developed hybrid system.

2.4 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 previously 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 [18] 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 [2, 4, 19, 33] 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 [30]. 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 [31]. 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 [2] 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 [15]. 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 [31]. A related problem is multicast group estimation [25], 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.

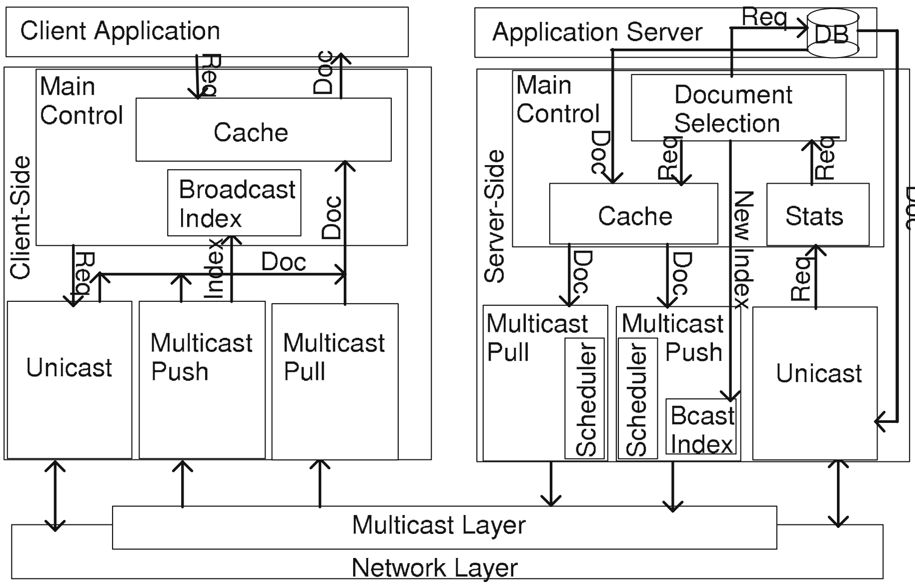

Figure 1 Our hybrid system.

3 Our hybrid system

In this section we describe our hybrid system and how it addresses the key issues about hybrid systems outlined in the previous section. In particular, we will be looking at the issue of mispredictions, shifting document popularities,document classification (deciding between what documents are hot and which are cold) and estimating document popularity for items on the push channel. We will address all of these issues below but for more information on our document selection method and coinciding experiments we refer the reader to our previous work [7].

We first present the configuration of our system, which is shown in Figure 1, for more information on this architecture please see [14, 26]. 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), which in the figure consists of both the Multicast layer and Network layer. This makes the identity of the multicast service not particularly relevant to our middleware. For our experiments in this paper we use Java Reliable Multicast Service (JRMS) [28] on top of TRAM multicast [13]. JRMS guarantees the reliable delivery of multicast packets. We maintain two multicast channels, one for pull and one for push.

3.1 Client description

In order to allow the clients to quickly determine whether a needed document 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 waits for item D_i to appear on the multicast push channel. If the index appears before D_i appears, go to step 1. Otherwise, D_i appears before the next index and 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 returns the data for D_i to the client side front end.
- 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 is possible that the server has just started to push the document. If the 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.

3.2 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. We chose to go with a FIFO based threshold division because of its ease of implementation and that it directly fits within our model. Other scheduling schemes have been implemented into our architecture, in particular the Longest Total Stretch First, however for this experiment we chose to go with the simplest scheme for experimentation.

Notice that what is accomplished by our system is scalability in cases where mispredictions and false popularity estimations occur. With the introduction of the third, multicast pull, channel, we have instituted a way to help out at the server when an influx of requests appears. This is the case when the wrong items are placed on the push channel, because the server will then be getting the high load of requests that exists for the popular items that should be on the push channel. As our experiments will show, the multicast pull channel, in the capacity we have used it, provides more robustness and better system performance when these different problems occur, thus improving the hybrid data dissemination model.

3.2.1 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 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.

4 Document classification and bandwidth division

Our proposed solution to document classification and bandwidth division is to use an integrated algorithm that minimizes average latency, called the selection-division (SELDIV) Algorithm. SELDIV simultaneously and to near-optimality solves the bandwidth division and document classification problems. To better understand the SELDIV algorithm, it is first necessary to understand the differences in average latency for the multicast push and for the unicast pull channels. 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 sent once, not fragmented, and broadcast sequentially. In particular, the delay expected on the push channel is equal to half the total time it takes to broadcast all documents placed on the channel, which given that document i is of size S_i, is $\frac{\sum S_i}{2}$ for all i on the push channel. This makes one of the goals to keep the amount of data on the push channel as small as possible while maintaining the pull channel below a full load level, as that will enable very fast response times for items on that channel, which helps to lower system latency.

The delays for pulled documents, however, are radically different from those of pushed documents. If document j is assigned to unicast pull, a client request for j is queued at the server for transmission. Let S_j be the size of document j. Basic queuing theory tells us that the corresponding queuing delay is either $O(S_j)$ 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 steps of our algorithm are shown in Algorithm 1. This solution works in conjunction with a subroutine shown as Algorithm 2. Algorithm 1 uses a tolerance factor $\epsilon > 0$, 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. The algorithm also assumes that the list of documents passed in is ordered by decreasing popularity (meaning item 1 is the most popular, item n the least popular) and that the list includes the popularity of the items on the push channel. The parameters for the algorithms are summarized in Table 2.

Algorithm 1 proceeds in the following manner. It first pre-computes the sums $\sum_i^k \lambda p_i S_i$ (which is a running total of the bandwidth requirements for the first k items) and $\sum_i^k S_i$ (which is a running total of the size of the first k items), placing the totals in the arrays rspt and sizeTotal respectively (Lines 1–4). This will help to optimize the run time because these values would otherwise have had to be computed

Table 2 Parameters for Algorithms 2 and 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

during every loop. The algorithm then sets the initial minimum latency to be 0 and maximum latency to be the amount of time required to send all the documents out over individual connections with each client (Lines 5–6). A binary search is then performed, for which each loop consists of taking the average latency between the current minimum and maximum latencies and passing that average latency to the subroutine in Algorithm 2. Algorithm 2 will use that average latency to calculate the number of items k that should be placed on the broadcast push channel (recall that the list of items is ordered by popularity, so this k refers to the k most popular documents)(Line 8–9).

Algorithm 1 SELDIV - Bandwidth Division and Document Classification

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

Using the return value of k most popular documents, Algorithm 1 calculates the amount of bandwidth that should be given to the pull channel based on the choice of k (Line 10). Notice that in this calculation, a value $\alpha > 1$ is used 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 a can also be thought of as a safety margin for the pull channel.

After calculating the pull bandwidth, the remaining bandwidth is partitioned to the push channel and it is determined whether the amount of push bandwidth provided is actually enough to sustain the amount needed for the push channel (Lines 11–16). If there is enough bandwidth, then the latency could be lowered, and the max latency is decreased and the search performed again. Likewise, if there is not enough bandwidth, the latency is increased and the search performed again. This continues until the ϵ value is met, at which point the number of items for the push channel (and therefore which items), the push channel bandwidth and the pull channel bandwidth are all returned to be used to divide the bandwidth and documents for the system.

Let us now examine the details of Algorithm 2. Algorithm 2 requires as input latency along with the request rate (λ), number of documents (n) and the list items with popularities p. It then calculates and returns k, the number of items that should be placed on the broadcast push channel. The starting point for Algorithm 2 is a method suggested by [6] 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.

Algorithm 2 operates by using two bandwidth costs, one if the item is kept on the pull channel and one if the items are placed on the push channel. 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. As it was stated in [6], a document should be pushed if $\lambda p_i S_i > S_i/L$. Because the items are passed in with an order of most popular (first item) to least popular (last item), a binary search can be performed on the items to find the item k at which the division should occur.

Algorithm 2 tryLatency

```
Require: $n$, $\lambda$, $L$, $p$ as defined in Table 2
Ensure: Returns the number $k$ of items pushed given that average latency of $L$ is
    required
 1: lMax ← $n$
 2: lMin ← 0
 3: while max − min > 1 do
 4:     $k$ ← (max + min)/2
 5:     if $(p_k \lambda L) > 1$ then
 6:         min ← $k$
 7:     else
 8:         max ← $k$
 9:     end if
10: end while
11: Return $k$
```

As the algorithms above show, we are able to determine, based on document popularity, request rates and available bandwidth, the correct division of documents for the push channel and the amount of bandwidth to provide to that push channel. This gives us a solution to the document selection and bandwidth division problems which exist for hybrid systems, allowing us to develop a fully integrated hybrid system for highly scalable data dissemination.

The running time of our solutions consists of several parts. First, we assume that the list of documents is ordered by popularity. The first time this list is created, it will take $O(n\log(n))$ to create the list, and $O(\log(n))$ to maintain the list thereafter. The actual run time of Algorithm 2 is $O(\log(n))$ for each run through with a different latency, and for algorithm 1 the runtime is $O\left(\log(\frac{\sum_{i=1}^{n} S_i}{B\epsilon})\right)$ for the number of loops through that binary search. Thus, the overall runtime is $O\left(\log\left(\frac{\sum_{i=1}^{n} S_i}{B\epsilon}\right)\log(n)\right)$.

5 Experimental methodology

The main evaluation metric is the client-perceived delays to download requested documents. Delay statistics incorporate the system's response times for requests and its resiliency to unexpected load peaks. The evaluation uses the complete and operational system described in Section 3. This middleware is also available for downloading and testing with any web server, following the terms of open source distribution, at http://db.cs.pitt.edu/software/mbdd/. However, we will also take appropriate steps to isolate the effects of multicast pull from ancillary and accidental factors that are present in such a complex system.

The emulation environment consists of a simulated application that uses the middleware and in which clients generate Poisson requests for documents. Clients and server run on the same machine to avoid network-induced variability and to isolate the intrinsic properties of multicast pull. This also allows us to slightly redefine the metric we use in our experiments. As mentioned above, the metric is the client-perceived delay to download requested documents. However, since the client and server are on the same machine, there is no real delay between the time a request is sent out of the server and the time it is received at the client. Therefore, it is possible to see this client-perceived delay as the time it took for the server to receive and respond to the request internally, but taking into account the time it takes to get information off of the multicast channels.

The emulation environment is a 2.0 GHz dual processor machine with 1.2GB of RAM and running Linux. In this paper, documents have fixed size: little deviations were found when documents have variable size and so the corresponding repetitive results are omitted. More generally, the correlation between document size and popularity is unclear [10] or weak and can be ignored [8]. A program was created to simulate the required request rate, which we refer to as the request filler. The request filler only added requests into the queue at the server if those items were not on the push channel, but it did not make explicit TCP connections with the server. The latency was measured at one client that actually made the connections with the server as a normal client would. On the whole, the client generated 10,000 requests during each experiment. Due to the nature of the client/server interaction, the feedback

Table 3 Simulation parameters.

Parameter	Value	Default
Document size	0.5K bytes	0.5K bytes
Zipf parameter θ	1.1–2.0	1.5
Multicast pull threshold	2	2
System bandwidth	100K bytes/s	100K bytes/s
Request rate λ	250/s	250/s
Re-configuration period	1–60 s	1 s
Total items n	100–10,000	1,000
Total requests made	10,000	10,000
Hot spot move type	Off, Small, Big	Off
α	1.1–4.2	2

mechanism was turned off during the experiments, forcing the measured client to make a request irregardless of the channel from which the document was retrieved.

The experimental parameters are found in Table 3 All parameters are fixed to the default value unless otherwise stated. The α parameter is a tunable factor in the bandwidth division algorithm and corresponds to the over-provisioning of the pull channel and was studied and explained in detail in previous work [7]. Specifically, the document selection component (Figure 1) assigns the document set between in the two categories of (unicast and multicast) pull and multicast push and it simultaneously partitions the server bandwidth between these two groups. The bandwidth division over-provisions the pull bandwidth by a factor of α given the currently predicted document popularity. In general, performance is fairly insensitive to the exact choice of α when the document access pattern is stationary (any α between 2 and 2.5 is acceptable [7]), but this is not necessarily the case when the client access pattern is not stationary.

Experiments were executed to evaluate non-stationary access patterns and the impact of multicast pull. In these experiment, the document popularity follows a Zipf distribution, i.e., the ith most popular document is requested with probability proportional to $1/i^{\theta}$. However, the exact identity of the ith most popular document changes with time. Changing popularity will be considered in the following two models. In the *small move* model, the popularities change gradually over time to reflect a gradual client shift in interest over time. In this model, 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. In the *big move* model, the location of the most popular document changes suddenly to reflect a sudden change in client interest, perhaps in response to an important event. The shift is simulated by making the probability of requesting the ith document proportional to $1/(1 + (i - b) \bmod n)^{\theta}$. The ordinary Zipf distribution corresponds to $b = 0$ and the document popularity can be quickly rearranged by changing the value of b. This means that by changing b to 1, documents will shift in popularity by one, so the most popular is no longer popular and the second most popular is now most popular, and so on.

6 Experiments

6.1 Experiment 1: document classification and bandwidth division evaluation

Figure 2 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 1 assigns the most popular seven documents on the push channel, and allocates 63% percent of the bandwidth to pull. 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 1 chose $k = 7$ point as the number of push documents, which is also the minimum point on the 65/35 curve. Thus, Algorithm 1 chose a better bandwidth split than the brute force approach and a document classification that was just as good.

6.2 Experiment 2: 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 3.

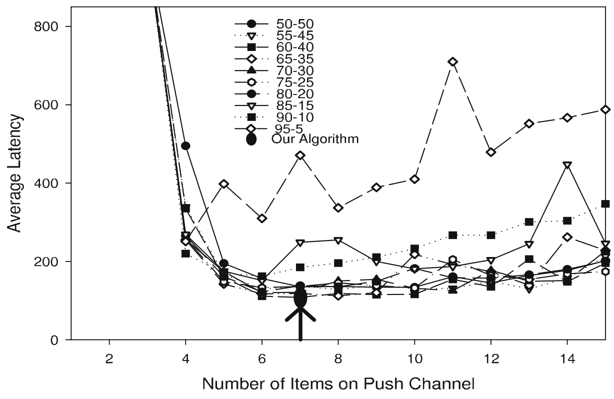

Figure 2 Optimality of algorithm 1 (*arrow* identifies single point found by the algorithm).

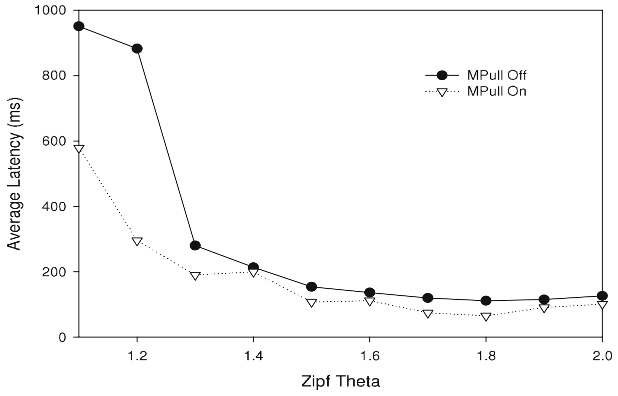

Figure 3 Average latencies for multicast pull On vs. multicast pull off for static access patterns.

Figure 3 shows the average latencies with multicast pull on, and 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 153.9 ms with multicast pull off, to 107.6 ms 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 4. For $\theta = 1.5$, the average latency with multicast pull off was 153.9 ms, with standard deviation of 308 ms. In contrast, with multicast on, not only was the average latency significantly better at 107.6 ms, but the standard deviation also improved to 226 ms.

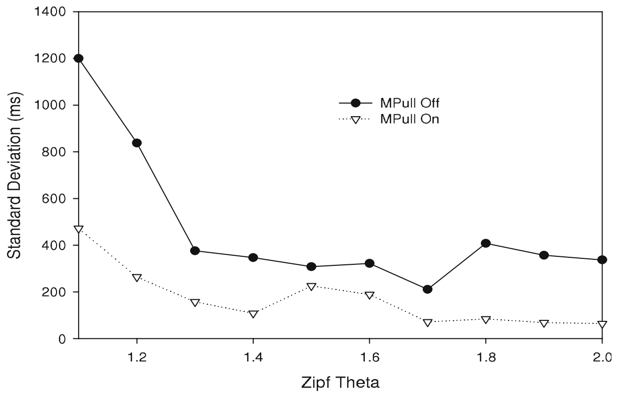

Figure 4 Standard deviations in latencies for multicast pull on vs. multicast pull off for static access patterns.

6.2.1 *Difference in latency of channel types*

One aspect we wanted to examine besides the average latency for the overall system was to examine the effects of the different hybrid system setups on the actual different channel types (push versus pull). Figure 5 shows the same setup and experiment as was shown in Figure 3, but this time the latency of the different channel types (push versus pull) have been separated instead of combined into a single overall latency. There are three main observations to make from this figure. The first is that the response times for the push channels are the same in both systems. This intuitively makes sense for the following reason. The number of items on the push channel, and which items are on the push channel, is the same in both hybrid systems. Therefore, the latency of that channel will be basically the same, because the same requests are being made for items on that same channel.

The second observation to make is that the difference in response times for items on the pull channels (unicast and multicast pull are considered the pull channels) is much larger than was shown in the overall latency graphs. Take the case of $\theta = 1.7$. In the overall latency graph, the difference between having multicast pull on or multicast pull off was only 15% but with latency looking at only the pull channel, the difference was almost 60%. The reason for this large difference is that the overall response time is dominated by response times for items on the push channel, and because of that the overall average latencies were driven low by the push average. With the response times for push items removed from the averages, the savings for pulled items is much higher and the usefulness of multicast pull it more apparent.

The final observation to make is that the savings for multicast pull on versus multicast pull off gets larger as the value of theta increases. The reason for this is that as theta gets larger, the number of items on the push channel is getting smaller, and the popularity of those items is getting larger. However, the popularity of the last few items left off the hot channel is also getting larger. Therefore, there will be more requests coming for items that are cold (those left off the smaller hot channel) and the server will have a larger load of overall requests but the similarity of those requests is larger. By having the multicast pull channel active, this load of similar

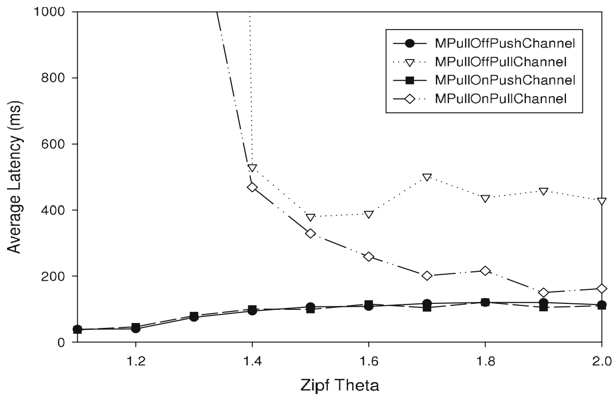

Figure 5 Difference in savings between channel types for multicast pull on vs. multicast pull off.

requests is more easily and efficiently handled, leading to lower average latency for items that are pulled.

6.3 Experiment 3: 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:

- 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.

The time between popularity shifting and the hybrid system adjusting to that change is not an instant process. There are several factors that can cause the system to be delayed in getting the appropriate documents split amongst the channel. One factor is the time delay between when the document classification was last run and is scheduled to be executed again. Another factor is the time it takes for the system to retrieve all the new documents for the push channel and begin to disseminate that data. Additionally, there is a delay that is incurred while the current broadcast cycle is being performed until the new broadcast cycle can be sent out. Finally, there is a delay that occurs from gathering the correct set of items to be put on the push channel based on the feedback mechanism in place and on the servicing of a sudden influx of requests for items. We show in this subsection that until the document classification is updated and reconfigured, multicast pull helps provide an intermediate form of scalability.

Figure 6 shows the resulting average latencies resulting from gradual changes in popularity while varying the θ and keeping other parameters fixed to their default value. In this experiment, 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

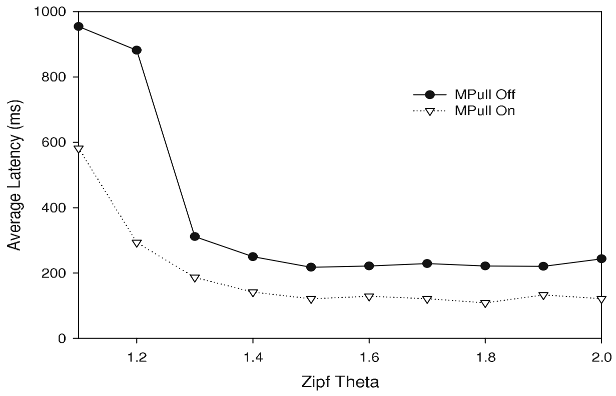

Figure 6 Average latency for multicast pull On vs. multicast pull off for small move access patterns.

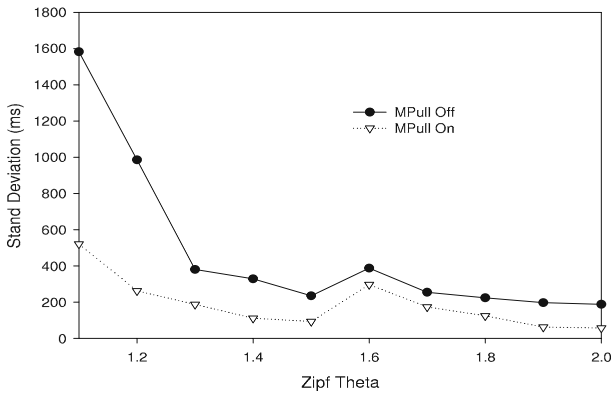

Figure 7 Standard deviation on latency for multicast pull on vs. multicast pull off for small move access patterns.

received from the monitored client. These 500 requests do not include requests made by the request filler.

Figure 6 shows that there is a tendency as theta increases for the benefit of the multicast pull channel to increase. This is especially true for low thetas ($\theta = 1.2$) versus high thetas ($\theta = 2.0$). This benefit can be seen, for example, When examining several different θ. When $\theta = 1.5$, multicast pull shows an improvement in average latency of 44.6% (from 217.9 to 120.9 ms), at $\theta = 1.7$ the improvement in average latency is 47.1% (from 228.9 to 121.1 ms), and at $\theta = 2.0$ the improvement in average latency is 50.3% (from 243.5 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.

Figure 7 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 235 to 93 ms. 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.

Figure 8 shows 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 this case, we again see that using

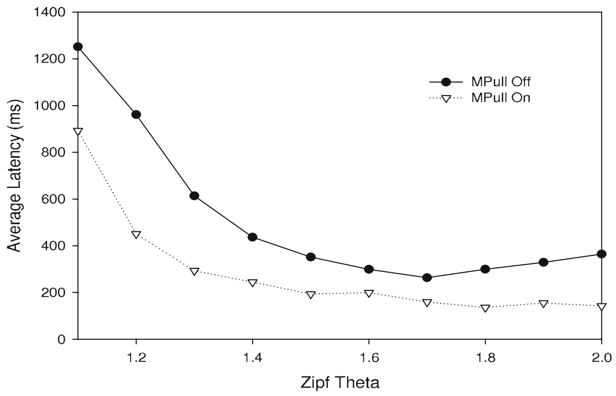

Figure 8 Average latencies for multicast pull on vs. multicast pull off for big move access patterns.

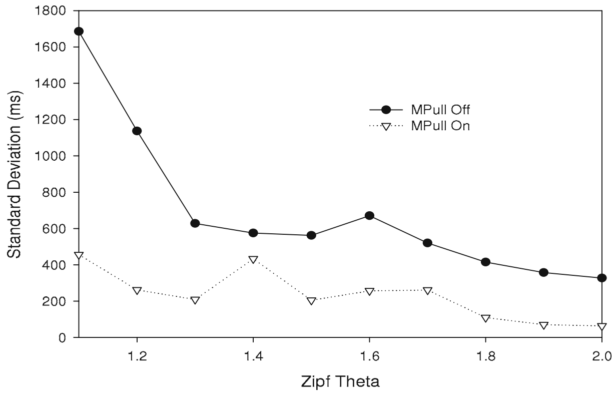

Figure 9 Standard deviations for multicast pull on vs. multicast pull off for big move access patterns.

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 average latencies. For $\theta = 1.5$ the reduction is 45% from 351.8 to 193.2 ms, for $\theta = 1.7$ the reduction is 40% from 263.4 to 159 ms, and for $\theta = 2$ the reduction is 61% from 364.7 to 141.8 ms.

Figure 9 shows that for big moves, multicast pull reduced the standard deviation of the latencies even more dramatically than for small moves. For $\theta = 1.5$ the standard deviation has decreased 63% from 562 to 205 ms, and for $\theta = 2$ the decrease was 81% from 327 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.

6.4 Experiment 4 - multicast pull advantage with varying time between reconfiguration

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.

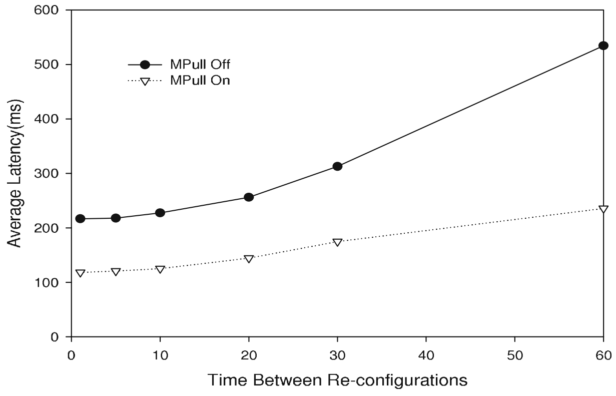

Figure 10 Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for small move access patterns.

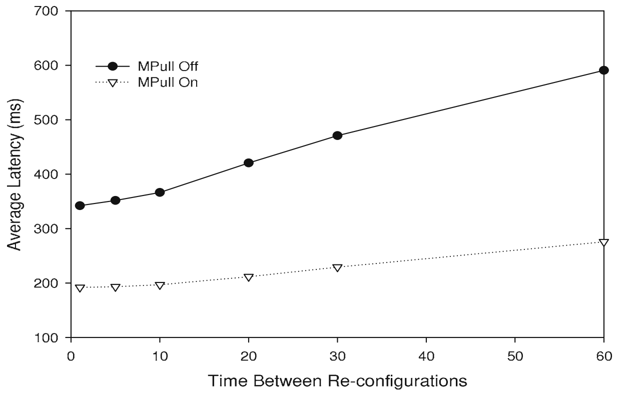

Figure 11 Multicast pull on vs. multicast pull off for varying reconfiguration times in seconds for big move access patterns.

Figures 10 and 11 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 s, 45% when the reconfiguration is every 20 s, and 55.6% when the reconfiguration is every 60 s.

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

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.

6.5 Simulated real world network scalability experiment

In this experiment, we changed the experimental environment to be in a real world environment where network latency exists, as do both the client and server systems experience operations besides those of our system. This means the systems are operating on other processes in addition to running our code, which adds in additional variability. In addition, we run these experiments with multiple clients, instead of using a single client with a request filler program. These clients are spread out across the globe, and thus vary in response times over all channels. The way this environment was created was by using the Planet Lab [27] network environment to run the experiments. For these experiments, the probability feedback (s_i) was set to be $2/(r_i)$ where r_i is the number of requests received for i during the current cycle. This would cause the server to receive approximately 2 requests for each popular document, instead of always forcing feedback as was the case in previous experiments.

We ran two separate experiments, one with static document popularity and one with the large document popularity moves mentioned in the previous experiments.

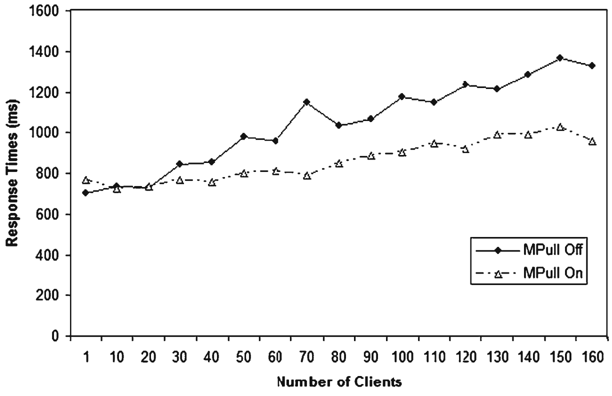

Figure 12 Multicast pull on vs. multicast pull off for real world experimental set-up and static access patterns.

In the experiments the number of clients was varied from 1 to 160 in the environment we mentioned. The response times were measured at each client in milliseconds and the average response times of all clients was combined to create the average response time that we report. The size of the documents in this experiment are varied from 0.5KB to 2MB in size, and are randomly spread out amongst all documents so there is no relationship between popularity and document size. Other relevant experimental parameters are that the zipf θ was set at 1.5, the default value.

The results of this experiment are found in Figures 12 and 13. Figure 12 show the results for the static document patterns for both the hybrid system with the multicast pull channel on and off. As the figure shows, having the militant pull turned on helped to keep the response times down, especially as the number of clients was increased. When the number of clients is low, most documents are responded to over unicast so the difference is mainly latency numbers, and not related to using the multicast pull channel or not. When the number of clients is 90, using multicast pull is 11.7% more effective, at 130 clients it is 13.8% more effective, and at 160 clients it is 21% more effective. The reason for this effectiveness is as the number of clients increases, there are more similar requests and the server can benefit from a single server send out.

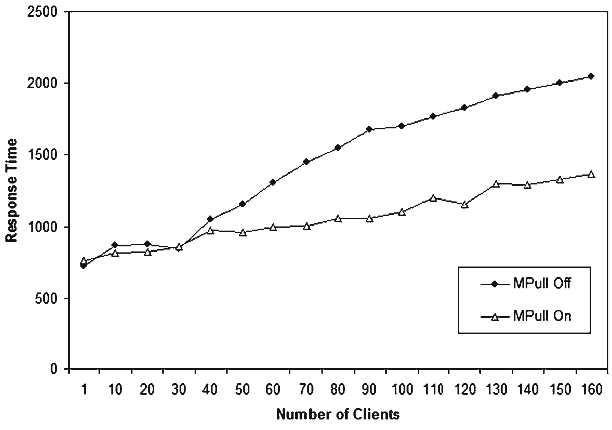

Figure 13 Multicast pull on vs. multicast pull off for real world experimental set-up and large move access patterns.

Also, because the server starts to experience a lot of request lag at around 40 clients, being able to have less requests at the server helps to speed up other requests for less popular items.

Figure 13 shows the same results as Figure 12 but for dynamic access patterns. This time, the benefit of using the multicast pull channel is even more prevalent. In this case, the multicast pull channel is 23.6% more effective in terms of response times for 60 clients, 37.1% more effective with 90 clients and 33.3% more effective with 160 clients. This follows the findings in our previous experimental set, where using the multicast pull channel was more effective than not using the multicast pull channel. Thus, we are able to show that the results in our confined simulation environment are similar in nature to those in a real world environment.

7 Conclusion

Scalability has been the number one problem in the efficient dissemination of data in the Internet. One technique that has shown much promise to achieve scalability is multicast push. More recently, hybrid systems that combine multicast push with unicast have been shown to provide scalability while keeping response times low. Given this ability to provide scalability at a minimal cost to the user, our contribution in this work focused on solving problems associated with the hybrid dissemination model. In particular, we (1) addressed the issues of *document popularity* and *document division* and (2) argued for the use of a third channel, called the *multicast pull channel*, in the hybrid system model. We also showed that this third channel is used to both help decrease latency and increase system robustness, especially in cases when document popularity is shifting or has been mispredicted.

Our experiments showed the quantification of the advantage of including a multicast pull component. We showed that using multicast pull is of modest, but measurably advantage at the level of 10–25% 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, where savings were found in the range of 44–55%. System variability was also shown to be lower, including decreasing by 40–60% when there are document popularity shifts. In such a dynamic environment, multicast pull provides an intermediate form of scalability until the document classification algorithm, which is one of the most expensive components in any hybrid system, is invoked.

Our overall research goal is to combine the hybrid data dissemination model with a peer-to-peer system that includes a built-in multicast data dissemination. Therefore, our current future work focuses on porting our system over a multicast capable peer-to-peer network, such as Scribe [11]. In such a system our middleware nodes will become peers acting as reverse proxies to multiple web/database servers. Middleware nodes will assist each other to deal both with data dissemination as well as dealing with failures.

Acknowledgements This research is supported in part by NSF grants CCR-0098752, ANI-0123705, ANI-0325353, and ANI-0123929. We would like to thank the anonymous reviewers for their very helpful comments.

References

1. Acharya, S., Muthukrishnan, S.: Scheduling on-demand broadcasts: new metrics and algorithms. In: Proceedings of the ACM/IEEE MobiCom, pp. 43–54 (1998)
2. Acharya, S., Franklin, M., Zdonik, S.: Balancing push and pull data broadcast. In: Proceedings of the ACM SIGMOD (1997)
3. Aksoy, D., Franklin, M.: Scheduling for large-scale on-demand data broadcasting. In: Proceedings of the INFOCOM, pp. 651–659 (1998)
4. Aksoy, D., Franklin, M.: RxW: A scheduling approach for large-scale on-demand data broadcast. ACM/IEEE Trans. Netw. **7**(6), 846–860 (1999)
5. Altinel, M., Aksoy, D., Baby, T., Franklin, M., Shapiro, W., Zdonik, S.: DBIS toolkit: adaptable middleware for large scale data delivery. In: Proceedings of the ACM SIGMOD (1999)
6. Azar, Y., Feder, M., Lubetzky, E., Rajwan, D., Shulman, N.: The multicast bandwidth advantage in serving a web site. In: Proceedings of the 3rd NGC, pp. 88–99 (2001)
7. Beaver, J., Morsillo, N., Pruhs, K., Chrysanthis, P.K., Liberatore, V.: Scalable dissemination: what's hot and what's not. In: Proceedings of the WebDB (2004)
8. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and Zipf-like distributions: evidence and implications. In: Proceedings of the INFOCOM (1999)
9. Burns, R., Rees, R., Long, D.: Efficient data distribution in a web server farm. IEEE Internet Computing **4**(5), 56–65 (2001)
10. Càceres, R., Douglis, F., Feldmann, A., Glass, G., Rabinovich, M.: Web proxy caching: the devil is in the details. In: Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance (1998)
11. Castro, M., Druschel, P., Kermarrec, A., Rowstron, A.: SCRIBE: a large-scale and decentralized application-level multicast infrastructure. IEEE J. Sel. Areas Commun. (2002)
12. Cheney slip sends Net surfers to anti-Bush site http://www.cnn.com/2004/ALLPOLITICS/10/06/debate.website.ap/index.html
13. Chiu, D., Hurst, S., Kadansky, M., Wesley, J.: TRAM: a tree-based reliable multicast protocol. Sun Microsystems, No. TR-98-66 (1998)
14. Chrysanthis, P.K., Pruhs, K., Liberatore, V.: Middleware support for multicast-based data dissemination: a working reality. In: Proceedings of the WORDS (2003)
15. Cormode, G., Muthukrishman, S.: What's hot and what's not: tracking frequent items dynamically. In: Proceedings of the Principles of Database Systems (2003)
16. Dykeman, H.D., Ammar, M., Wong, J.W.: Scheduling algorithms for videotex systems under broadcast delivery. In: Proceedings of the International Conference on Communications, pp. 1847–1851 (1986)
17. Foltz, K., Xu, L., Bruck, J.: Scheduling for efficient data broadcast over two channels. In: Proceedings of the ISIT (2004)
18. Franklin, M., Zdonik, S.: Data in your face: push technology in perspective. In: Proceedings of the AMC SIGMOD (1998)
19. Hall, A., Taubig, H.: Comparing push- and pull-based broadcasting. Or: would "microsoft watches" profit from a transmitter? Lect. Notes Comput. Sci. **2647**, 622 (2003)
20. Jannotti, J., Gifford, D., Johnson, K., Kaashoek, M., O'Toole Jr., J.: Overcast: reliable multicasting with an overlay network. In: Proceedings of the OSDI, pp. 197–212 (2000)
21. Kenyon, C., Schabanel, N.: The data broadcast problem with non-uniform transmission times. In: Proceedings of the SODA, pp. 547–556 (1999)
22. Kenyon, C., Schabanel, N., Young, N.: Polynomial-time approximation scheme for data broadcast. In: Proceedings of the STOC, pp. 659–666 (2000)
23. Kleinrock, L.: Queueing Systems, Vol. 1: Theory. John Wiley & Sons (1975)
24. Krishnamurthy, B., Wills, C., Zhang, Y.: On the use and performance of content distribution networks. In: Proceedings of the ACM SIGCOMM Workshop on Internet Measurement, pp. 169–182 (2001)
25. Nonnenmacher, J., Biersack, E.: Scalable feedback for large groups. IEEE/ACM Trans. Netw. **7**(3), 375–386 (1999)
26. Penkrot, V., Beaver, J., Sharaf, M.A., Roychowdhury, S., Li, W., Zhang, W., Chrysanthis, P.K., Pruhs, K., Liberatore, V.: An optimized multicast-based data dissemination middleware: a demonstration. In: Proceedings of the ICDE (2003)
27. PlanetLab—An open platform for developing, deploying, and accessing planetary-scale services, http://www.planet-lab.org/

28. Rosenzweig, P., Kadansky, M., Hanna, S.: The java reliable multicast service: a reliable multicast library. Sun Microsystems, No. SMLI TR-98-68 (1998)
29. SpaceShipOne captures X Prize http://www.cnn.com/2004/TECH/space/10/04/spaceshipone.attempt.cnn/
30. Stathatos, K., Roussopoulos, N., Baras, J.S.: Adaptive data broadcasting using air-cache. In: Proceedings of the WOSBIS (1996)
31. Stathatos, K., Roussopoulos, N., Baras, J.S.: Adaptive data broadcast in hybrid networks. In: Proceedings of the VLDB, pp. 326–335 (1997)
32. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, J.: Chord: ascalable peer-tp-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM, pp. 149–160 (2001)
33. Triantafillou, P., Harpantidou, R., Paterakis, M.: High performance data broadcasting systems. Mob. Netw. Appl. **7**, 279–290 (2002)
34. Zhang, W., Li, W., Liberatore, V.: Application-perceived multicast push performance. In: Proceedings of the IPDPS (2004)