Scalable Data Dissemination Using Hybrid Methods *

Wenhui Zhang, Vincenzo Liberatore
Division of Computer Science, EECS
Case Western Reserve University, Cleveland, Ohio 44106
E-mail: {wxz24, vxl11}@case.edu

Jonathan Beaver, Panos K. Chrysanthis, Kirk Pruhs
Dept. of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260
E-mail: {beaver, panos, kirk}@cs.pitt.edu

Abstract

Web server scalability can be greatly enhanced via hybrid data dissemination *methods that use both unicast and multicast. Hybrid data dissemination is particularly promising due to the development of effective end-to-end multicast methods and tools. Hybrid data dissemination critically relies on* document selection *which determines the data transfer method that is most appropriate for each data item. In this paper, we study document selection with a special focus on actual end-point implementations and Internet network conditions. We individuate special challenges such as scalable and robust popularity estimation, appropriate classification of hot and cold documents, and unpopular large documents. We propose solutions to these problems, integrate them in MBDD (middleware support multicast-based data dissemination) and evaluate them on PlanetLab with collected traces. Results show that the multicast server can effectively adapt to dynamic environments and is substantially more scalable than traditional Web servers. Our work is a significant contribution to building practical hybrid data dissemination services.*

1 Introduction

Web server scalability can be greatly enhanced with *hybrid data dissemination* that uses both unicast and multicast [7, 11, 19, 28]. Hybrid data dissemination is particularly promising due to the development of effective end-to-end multicast methods and tools [8, 9, 15, 17, 20, 22]. Hybrid data dissemination critically relies on *document selection*, which determines the data transfer method that is most appropriate for each data item. In this paper, we study document selection with a special focus on actual end-point implementations and Internet network conditions. We individuate special challenges such as scalable and robust popularity estimation, appropriate classification of hot and cold documents, and unpopular large documents. We propose solutions to these problems, integrate them in MBDD (middleware support multicast-based data dissemination) and evaluate them on PlanetLab [1] with collected traces.

Hybrid data dissemination uses a combination of unicast and multicast channels. Multicast channels are used for *hot* (popular) documents that are of interest to most clients. Multicast aggregates the servicing of hot documents into a single transmission and is thus more scalable than traditional unicast. However, multicast is inappropriate for *cold* (unpopular) documents since the multicast channel would force unwanted contents on most clients. Therefore, cold documents use traditional unicast. As a result, hybrid data delivery would ideally achieve scalable utilization of server and network resources while avoiding the reception of unneeded contents. However, the ideal picture is contingent on several assumptions, and in particular on the server's ability to partition documents as hot or cold. Document selection determines the data transfer method that is most appropriate for each data item. While previous work often focuses on one or several separate issues involved in document selection, this paper attempts to investigate these issues together, and devise integrated solutions for the document selection problem.

The first issue in document selection is to estimate document popularity. The problem can be further divided into two aspects. (1) *Popularity of cold data*. References to cold documents can be recorded during the

*This research is supported in part under NSF grant ANI-0123929.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

processing of client requests. The challenge lies in monitoring the unicast workload and collecting popularity information in the presence of workload dynamics. For example, if unicast connections are rejected at the operating system level due to unexpected sudden increase of unicast workload, the server can underrate the popularity of cold data since it fails to see the denied connections. In this paper, we present a two-phase processing design for accommodating client requests and capturing workload dynamics. Our design enables the server to adapt to a rapidly changing environment. (2) *Popularity of hot data*. Accesses to hot data are not seen by the server. One effective method to evaluate popularity of hot data is to sample clients, where the server multicasts polling requests so that clients would submit reports of references to hot documents with a specified probability. In practice, some concerns arise. For example, packets for polling requests can fail to reach a portion of clients if the underlying multicast protocol is not reliable [8], which leads the server to underestimate the popularity of hot data. We provide a scalable sampling approach that is highly resilient to packet loss.

The second key problem in document selection is the algorithm for classifying hot documents given popularity estimates. Previous algorithms are designed with the assumption that multicast bandwidth can be set to any amount within the total server bandwidth [7, 11]. However, in the Internet, the feasible multicast bandwidth is generally small compared with the available server bandwidth [27]. Hence, the problem needs to be reexamined. In this paper, we propose a simple and efficient algorithm for hot document selection. The algorithm ensures scalability by controlling the server perceived workload. Unlike early algorithms that assume a flat schedule on the multicast push channel [7, 11, 28], our algorithm allows the server to select more advanced scheduling strategies.

Another major problem is the selection of *warm documents* (defined in Sec. 2). Warm documents can solicit a big number of client requests and cause a sudden load increase on the server. One solution is to employ the multicast pull technology, where the server broadcasts warm data on an additional multicast channel (the multicast pull channel) [13]. However, previous work has not adequately investigated the selection method of warm documents. In this paper, we argue that warm document selection should be conservative to improve performance. As a consequence, we propose a *lazy warm document selection* scheme, which delays warm document selection until there is a potential of server overload.

In this paper, for the first time, we demonstrate that unpopular large documents notably impair performance. We give an efficient method to erase their negative effects by improving the hot document selection algorithm.

We implement our solutions in the context of the MBDD architecture [13], and evaluate them on PlanetLab (an Internet based network testbed) using collected web traces. The trace-driven experiments show that our multicast server is scalable and employs significantly less bandwidth than traditional web servers to provide comparable services. The multicast server is capable of controlling the incoming rate of unicast requests in the presence of traffic dynamics, thereby avoiding overloads. Furthermore, we conduct synthetic experiments on PlanetLab, which demonstrate the fast speed of the multicast server to adapt to huge workload increase by means of the document selection algorithms.

The remaining paper is structured as follows. Sec. 2 describes the background of MBDD. In Sec. 3, we investigate document selection and propose our solutions. Sec. 4, 5 and 6 report our experiments. In Sec. 7, we further compare our work with prior related work. Sec. 8 concludes the paper.

2 Background: The MBDD Middleware

The MBDD middleware unifies data management methods and data communication techniques into a software distribution [13]. It frees distributed applications from the details of underlying multicast transport and provides them with a scalable data management layer.

Data dissemination methods: The middleware utilizes three data dissemination methods: multicast push, multicast pull and unicast pull. In *multicast push*, the server repeatedly broadcasts documents (data items identified by URIs, e.g., an HTML file, an image file) to clients. Clients wait on the multicast channel for the requested data without sending explicit requests to the server. Multicast push is ideal for propagating hot documents. In *multicast pull*, clients submit explicit requests for documents and the server multicasts the requested data to clients. When multiple clients access the same document within a short time, the server multicasts the document only once. Multicast pull is suitable for disseminating *warm* documents, for which repetitive multicast push cannot be justified, while there is an advantage in aggregating concurrent client requests. Traditional unicast pull is preserved to disseminate cold documents.

Architecture: The MBDD server comprises of a number of components. The document selection component collects statistics on document popularity. The component periodically performs document selection to update the hot document group. Each selection cycle is called a document selection period (a *DS period*). An index of current hot documents is constantly broadcast on multicast push so that clients can quickly find out if

a document is on the multicast push channel. Unlike hot document selection, warm document selection can take place any time as needed (see Sec. 3.4). The multicast push scheduling component determines the frequency and order by which hot documents are broadcast. The multicast pull scheduling component arranges the sequence in which warm documents are disseminated.

To acquire a hot document, an MBDD client simply waits for the data on the multicast channels. To obtain a non-hot document, the client opens a TCP connection to the server and submits a request over it; then the client waits for the data on both the unicast channel and the multicast channels.

At both client and server sides, the transport adaption layer (TAL) allows the middleware to interact with various underlying multicast overlays with a uniform API. Although large scale overlays are still an active research area, a discussion of the underlying TAL overlay is out of the scope of the paper.

Supporting web data dissemination: Web applications interact with the middleware using the HTTP protocol. The MBDD client sits under web clients such as browsers. Web clients send HTTP requests to the MBDD client and receive HTTP responses from it. The MBDD server sits in front of a back-end web server. The MBDD server obtains the content of documents from the web server using HTTP. In [31], we give more details and provide the method to incrementally deploy MBDD in the Internet.

3 Document Selection Issues

3.1 Popularity of Non-hot Documents

3.1.1 Processing Unicast Requests

Clients access non-hot documents by making unicast requests. Hence, the problem of evaluating popularity of non-hot data can be studied by examining the processing of client requests at the server.

Fig. 1(a) describes a design that processes a unicast request in two phases. In the first phase, the Parser accepts a request, parses it, and reports the targeted document D to the DocSelection which is in charge of document selection. Then, the request is added to a first-in-first-out (FIFO) queue named the RequestQueue. The DocSelection increases the count of references to D by 1. Meanwhile, the DocSelection analyzes the current workload and may make a warm document selection (see Sec. 3.4). If a warm document W is successfully selected, all requests for W in the RequestQueue are removed and dropped since W will be broadcast on the multicast pull channel. Dropping a request leads the server to close the corresponding unicast connection. As a result, requests for warm documents do not need the second phase processing.

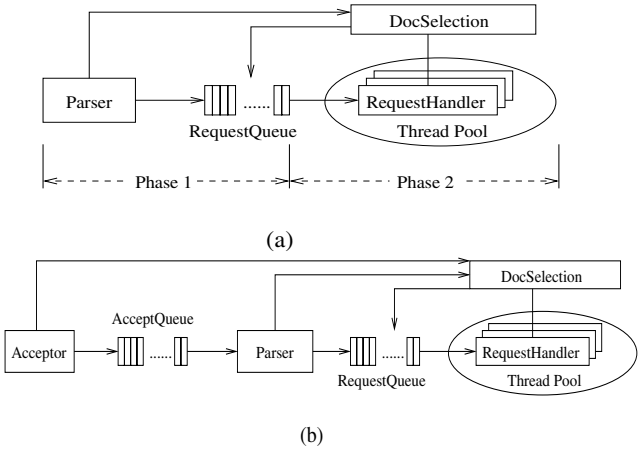

Figure 1. Two-phase Processing

The second phase is handled by RequestHandlers in a thread pool. A RequestHandler takes a request from the RequestQueue, and sends back the requested data over the unicast connection associated with the request.

The RequestQueue's size is bounded. This helps to keep requests served within a reasonable time. When the queue size reaches the uplimit and no warm document can be selected to reduce it, any new parsed request must be dropped. However, before the drop, the victim request is reported to the DocSelection, so that no popularity information is lost.

In this design, the server uses the Parser to monitor two things. First, unicast workload is observed by counting the total unicast references R. Second, the access probability of a document by unicast requests is computed as r/R, where r is the number of references to the document.

3.1.2 Improving Adaptability

The server can perceive a large and rapid increment of unicast requests. For example, several cold documents suddenly become extremely popular. If the rate of incoming requests exceeds the processing speed of the Parser, a portion of client connection requests can be silently rejected at the operating system level. Being blind to the denied connections, the server underestimates the workload and is unable to perform appropriate document selection to adapt promptly. We show this fact with experiments in [31].

Fig. 1(b) shows an improved design to enhance server's adaptability. The basic idea is to separate workload monitoring from the Parser. We add the Acceptor to accept requests and report workload. Upon accepting a

request, the Acceptor asks the DocSelection to increase a counter that records the total unicast accesses. The accepted request is then put in an FIFO queue called the AcceptQueue. The Parser takes requests from the AcceptQueue and parses them. As the Acceptor's task is light, arrived requests are quickly accepted and counted, thereby avoiding having requests accumulated at the operating system level. In a DS period, if the Acceptor accepts n requests, and among m requests parsed by the Parser, r of them refer to document D, the DocSelection assumes there are nr/m requests for the document. If the DS period length is L, the access rate of D is evaluated as nr/mL.

The AcceptQueue's size is bounded to avoid a huge number of concurrent connections that consume excessive resources. Should the queue be full, the Acceptor drops newly accepted requests after reporting their arrivals to the DocSelection. As a result, the server is kept aware of the actual incoming request rate, which enables it to conduct appropriate document selections.

3.2 Popularity of Hot Documents

To collect hot data popularity, the server multicasts a polling request (PR) that asks clients to submit reports with probability ρ. A report lists the hot documents accessed during a time interval with their reference counts. A major concern is that a PR can fail to reach some clients if multicast is not reliable [8]. We solve the problem by having the server broadcast the PR multiple times in each DS period. To achieve scalability, our approach spreads the arrivals of reports at the server to prevent report bursts.

Suppose a DS period with identifier I starts at time t_0 and ends at time t_1. The server creates a PR at t_0 and transmits it on multicast push every α seconds during the period. Furthermore, the server expects to receive client reports in the interval of $[t_1 - \beta, t_1)$. By tuning β, the server can spread out the report arrivals. Every time the PR is transmitted, it contains I, ρ, β and ζ, where ζ is the length of time between the current transmission and t_1. For PRs of the same DS period, a client only reacts to the first one that is received, and it replies to it with probability ρ. If $\beta \leq \zeta$, the client should reply between time $t_1 - \beta$ and time t_1, and in practice it counts accesses to hot documents for $\zeta - \gamma$ seconds, where γ is a random value in $(0, \beta]$, and submits a report. If $\beta > \zeta$, the client has received the PR during the expected reply interval, and so it computes statistics for γ seconds before submitting a report, where γ is a random value in $[0, \zeta)$. Note the length of the sampling interval is piggybacked to the report.

The server defines a valid report as one that was collected for an interval no less than a threshold (we simply set it to α in our experiments). Suppose among m received reports, k of them are valid. Further, suppose the access rate to a document in report i is λ_i, the estimated access rate of the document is calculated as $m\Sigma_{i=1}^{k}\lambda_i/\rho k$. This popularity estimate can be analyzed in terms of its accuracy. However, in this paper, we omit the analysis except to observe that the server is robust to sampling errors thanks to the warm document selection (Sec. 3.4).

The reports must be processed in time for effective new hot document selection. The server assigns higher priority to reports over any other requests. Handling a report is efficient for it does not incur the overhead associated with a response. Moreover, in MBDD, identifiers of documents are represented with digests [24]; thus, the reports are quite compact.

The server controls the number of reports by specifying a proper ρ. Suppose μ is the intended number of reports in a DS period, n is the estimated number of clients, then $\rho = \mu/n$. The actual number of received reports is used to adjust the value of n, which in turn is employed to decide ρ for the next DS period. Effective methods can be adopted for optimal estimation of n [5].

3.3 Hot Document Selection

Hot documents are selected at the beginning of a DS period based on data popularity in the previous DS period. Algorithms in earlier research [7, 11, 28] make one or both of the following two assumptions. First, the multicast bandwidth can be set to any amount within the total available server bandwidth. Second, the multicast scheduling is simply flat (*flat multicast*), where each document is sent exactly once in every multicast period.

In the Internet, a feasible multicast bandwidth is generally small compared with the server bandwidth. This is because the multicast bandwidth of end-to-end multicast networks is restricted by the limited uplink bandwidth of the participating end hosts [27]. In most DSL and cable connections, a host's outgoing bandwidth is configured to be much smaller than incoming bandwidth. Hence, in this paper, we model the problem as document selection given a small fixed multicast bandwidth. Constant multicast bandwidth is also used in [23, 28].

To achieve high scalability, the primary objective of document selection is to guarantee the server perceived workload (unicast requests for non-hot documents) within its service capacity. The workload is reduced when assigning more popular documents to the multicast push channel. However, when the server is not overloaded, unicast communication is preferred to transmit a document since it results in smaller client perceived latencies. Therefore, the essential idea of our algorithm is to assign as many documents as possible to

the unicast channel, while guaranteeing the server's scalability by keeping the server workload under capacity.

As in [25], the server capacity is determined by two factors: (1) the local resources of the hosting machine, and (2) its bandwidth. First, the CPU cycles and memory size restrict the number of connections simultaneously held by the server, thereby putting a cap on the maximum request rate that can be served on unicast. For our analysis, the server processing capacity is expressed as a target number R of unicast requests per second. Second, a bandwidth bottleneck is modeled as server's uplink bandwidth constraint as in [25]. When the multicast bandwidth is constant, the uplink bandwidth for unicast is also fixed. We let U be a parameter that represents the target unicast bandwidth. Its value should be below the actual maximum unicast bandwidth. It is otherwise a tunable parameter.

Suppose λ_i is the access rate to document i, $i = 1, ..., n$. Let Set_u be the collection of documents assigned to the unicast channel. To control requests for non-hot data, we require

$$\Sigma_{i \in Set_u} \lambda_i \leq R. \quad (1)$$

Define the size of document i as s_i. To prevent sever overload, we must keep the bandwidth consumption below U, hence,

$$\Sigma_{i \in Set_u} \lambda_i s_i \leq U. \quad (2)$$

Algorithm 1 outputs the set of hot documents. The documents are sorted by their popularity in ascending order. If two documents have the same popularity, the one with larger size will sit before the other in the ordered sequence. The algorithm scans the documents from least to most popular and puts them to the unicast channel as long as the above inequalities (1) and (2) still hold. Note that a selected hot document by the algorithm can have a smaller access rate than a cold document. The algorithm is both simple and efficient. Its running time is $O(n \log n)$. We will improve the algorithm in Sec. 5.5.

When the sending rate is constant, the multicast push performance mainly depends on two factors: the scheduling and the size of hot data. Our algorithm does not specify a multicast schedule, hence, it allows the server to employ sophisticated scheduling strategies [10, 29]. To deflate the size of hot documents, compression technology can be adopted.

Note given a multicast bandwidth and multicast schedule, as the number of clients scales up, the latency to wait on the multicast channel increases since the server selects more hot documents. However, the latency still compares favorably with the infinite latency of an overloaded server.

Algorithm 1 Hot Document Selection

Require: $U, R, \lambda_1, \lambda_2, ..., \lambda_n, s_1, s_2, ..., s_n$
1: $Set_{hot} \leftarrow$ empty
2: $i \leftarrow 1, r \leftarrow 0, bw \leftarrow 0$
3: Sort the documents so that $\lambda_{k_1} \leq \lambda_{k_2} \leq ... \leq \lambda_{k_n}$, and if $\lambda_{k_j} = \lambda_{k_{j+1}}$ then $s_{k_j} \geq s_{k_{j+1}}$, where $1 \leq j < n$
4: **while** $(i \leq n)$ **do**
5: **if** $(r + \lambda_{k_i} > R)$ **then**
6: Add document $k_i, k_{i+1}, ..., n$ to Set_{hot}
7: Return Set_{hot}
8: **else if** $(bw + \lambda_{k_i} s_{k_i} \leq U)$ **then**
9: $r \leftarrow r + \lambda_{k_i}, bw \leftarrow bw + \lambda_{k_i} s_{k_i}$
10: **else**
11: Add document k_i to Set_{hot}
12: **end if**
13: $i \leftarrow i + 1$
14: **end while**
15: Return Set_{hot}

3.4 Warm Document Selection

Warm document selection examines two questions: how and when to perform the selection. If multicast can use any fraction of the available server bandwidth, the selection can be triggered as long as there are multiple outstanding unicast requests for one document [12]. By aggregating requests, the server reduces the workload and clients experience less latency. To select warm documents, the server counts the outstanding requests on each document; if the counted number reaches a threshold, called the *warm threshold*, the corresponding document is classified as warm and scheduled on the multicast pull channel; meanwhile, outstanding unicast requests on the document are dropped.

However, in the Internet, where the feasible multicast bandwidth is small, multicast pull competes with multicast push for the limited multicast bandwidth. On one side, the performance of multicast push is negatively affected by contention with multicast pull. For example, with a flat schedule, the mean latency perceived on multicast push is inversely proportional to the bandwidth available to the push channel. As the workload turns heavier, the impact on overall performance becomes more evident because an increased fraction of all requests are satisfied by the multicast push channel. On the other hand, if we schedule a document on multicast pull every time the warm threshold is reached, warm documents can experience long waiting time before being multicast due to a large volume of warm data.

In this paper, we propose a *lazy warm document selection* (LWDS) scheme, where the server triggers warm document selections only if it sees a potential of over-

load. We define a threshold T that is no more than the maximum size of the RequestQueue. A warm document selection is *only* performed when the RequestQueue's size exceeds T. If warm documents exist, *only* the most popular one is scheduled on multicast pull, and any request for the document is removed from the RequestQueue so that the queue size is reduced. In case the RequestQueue is full and no warm document can be selected, newly parsed requests will be dropped by the Parser. To send a warm document, the multicast pull channel preempts a fraction of multicast bandwidth from multicast push. The bandwidth is returned to multicast push when no document is scheduled on multicast pull.

In LWDS, the primary goal of warm document selection is to relieve a sudden and heavy load, thereby ensuring server scalability. Instant heavy workload can be caused either by workload dynamics or by the fact that a popular document is incorrectly classified as cold. Moreover, warm document selection also makes the server robust to the setting of the parameter U in Sec. 3.3. In our experiment, U is simply set to the maximum available unicast bandwidth.

4 Experimental Methodology

We implement the proposed solutions in Java in the context of MBDD and evaluate them on PlanetLab. We conduct two types of experiments: trace-driven experiments and synthetic experiments. The trace-driven experiments aim at demonstrating the significant scalability enhancement by MBDD in handling real web server workload compared with traditional unicast servers. The first metric is the client perceived latency which is defined as the time between a document request is generated and the request is satisfied. The second metric is the ratio of the bandwidth required by an MBDD server and by a unicast server to accommodate the same workload. Our synthetic experiments stress the speed of server adaptation under rapid and large traffic increase, and also emphasize the effectiveness of our warm document selection scheme in MBDD from the perspective of the client perceived latency. For lack of space, we only highlight the results of our synthetic experiments in Section 6; details can be found in [31].

4.1 Trace Extraction

The Soccer World Cup 1998 server trace is one of the busiest recorded so far, and is extensively and currently used in research [25, 26, 30]. During peak time, the 30 World Cup servers receive more than 10 million requests per hour [6]. We extract our experimental trace from the original trace between 6pm and 11pm (coord.univ.time) on July 8th. The interval includes the semi-final game between France and Croatia [1] (7pm-9pm). The selected trace exhibits a typical server workload behavior during the course of a hot event: the traffic volume increases tremendously after the event begins, and falls back to normal level at the end of the event.

Table 1. Trace Information

Trace	#Client	#Request	#File	File Size (byte)		
				Mean	Median	Max
Original	69013	21254172	11312	14867	5620	2.89M
Experiment	420	3665066	9042	12624	5659	2.89M

To fit the trace into PlanetLab scale, we filter out most clients from the original trace. The experimental trace comprises of the busiest 420 client traces. We keep the requests that are GET or HEAD methods and result in 200 (ok) and 304 (not modified) responses. These requests account for about $1/6$ of the original workload. The top rate is as high as 700 req/sec (requests per second). The number of requests in each client trace ranges from 4386 to 43687. Table 1 presents the statistics of the original trace and the experimental trace.

The experimental trace captures most of the major workload characteristics of the original trace. Fig. 2(a), (b) and (c) compare the original and experimental traces in their distributions of file popularity, file sizes and response sizes respectively. The two traces follow similar distributions except that the file popularity distribution of the experimental trace is slightly more skewed. Fig. 2(d) compares the workload variation over time, where the request rate is normalized to the maximum rate. The request pattern of the experimental trace closely matches that of the original trace during the game. Although there is a difference before and after the game, it does not matter much since our purpose is to demonstrate our server's scalability under heavy workloads. When the workload is light, the server behaves like a unicast web server.

4.2 Experiment Setup

We conduct our experiments on PlanetLab, which allows us to investigate the system under real Internet conditions and at large scale. We utilize 210 machines distributed across North America and Europe. As the resource of a PlanetLab node is often shared by several tens of slices [16], we assign only 2 virtual clients to a machine to avoid overloading the node. Each of these 420 virtual clients runs as an MBDD client and creates its own workload independently based on a distinct client trace. We also set a timeout of 120 seconds for

[1] We do not select the final game because its trace is quite light. The game was held on Sunday, thus, most people watched it on TV [6].

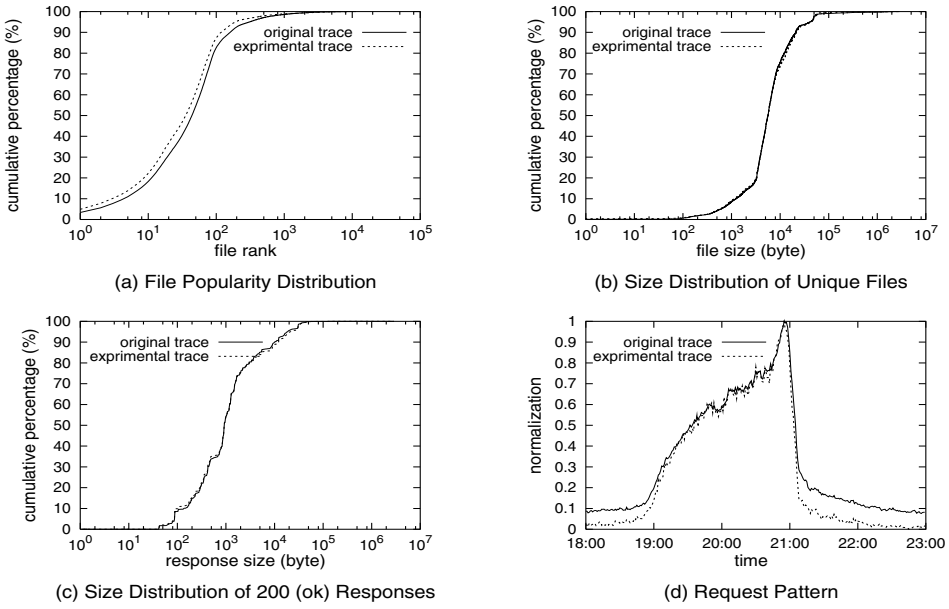

Figure 2. Comparison of Original and Experimental Workload

requests waiting on multicast channels to avoid a high number of cumulative concurrent requests that consume excessive system resource. Moreover, if a request waiting on the unicast channel receives no data in any period of 120 seconds, it will be aborted and logged as a timeout request. We will quantify the impact of timeout requests. As the middleware runs at both client and server sides, it is easy to employ compression technology. A typical factor of gain in HTML file size is more than 3 [18]. Since contents of documents are not available, we assume the size of an HTML file is reduced by two thirds.

The MBDD server is on a non-PlanetLab machine in our lab. The machine runs Linux Fedora 2 and is equipped with a 1.7G Intel Pentium 4 processor and 512 RAM. We limit server's unicast uplink bandwidth to 1.6Mbps so that the server will employ multicast communication under the experimental workload. When no warm data are on multicast pull, the multicast push rate is 160Kbps; otherwise, the sending rates on multicast pull and multicast push are 40Kbps and 120Kbps respectively. The scheduling of multicast push is MAD [29] and that of multicast pull is LTSF [3]. Each multicast channel runs on a simple end-to-end multicast overlay network. The topology of the multicast overlay network is a binary tree rooted at the server. Data delivery between nodes employs TCP connections. A document is broken into 1024-byte chunks to multicast on the network. Each end node has a buffer of 50 chunks, hence, data can be dropped out at the node due to temporary network congestion.

The server has 50 RequestHandlers. Both the AcceptQueue and RequestQueue have an uplimit of 100 requests. Warm document selection is triggered when the RequestQueue's size is 100. The warm threshold is set to 4.

The server makes hot document selection every 60 seconds. In the selection algorithm, the threshold parameters U and R are set to 200KBps (1.6Mbps) and 100 req/sec respectively. The server samples clients to collect popularity of hot documents. The targeted number of reports is set to 40 in a DS period; and $\alpha = \beta = 5$ seconds.

5 Evaluation

5.1 Overall Performance

We leave out the results where only the unicast channel is active because the MBDD server acts as a traditional unicast server at the time. We observe the server employs both unicast and multicast during the period of emulating the trace with log time between 7:00pm and 9:15pm. In this examined interval, clients generate a total of 3213190 requests [2] toward the server and

[2] Should the same workload be satisfied by a unicast server with the same bandwidth, the average latency is 4993 seconds; this simulation assumes the server has an infinite waiting queue and can hold unlimited unicast connections, and an HTML file is compressed to 1/3 of its original size.

Table 2. #Request and Average Latency

	#Request	%	Ave.Lat.
Overall	3213190	100	2795 ms
Multicast Push	2400475	74.7	3338 ms
Multicast Pull	51354	1.6	4815 ms
Unicast	749583	23.3	931 ms

Table 3. #Request and Average Latency at Peak Time

	#Request	%	Ave.Lat.
Overall	432197	100	3985 ms
Multicast Push	374345	86.6	4349 ms
Multicast Pull	5225	1.2	3588 ms
Unicast	51828	12.0	1392 ms

99.6% of them receive the requested data successfully. The remaining 0.4% of requests are aborted due to timeouts. As timeouts are rare, they have no significant effect on the performance. Table 2 displays the number of requests served by each channel and the average latencies in millisecond. The latency is the time between a request is generated and the request is satisfied by a channel. The third column of Table 2 shows the percentage of requests served by each channel. The table indicates multicast push satisfies most of all requests while keeping the mean latency within a reasonable amount. Unicast responses are delivered at a high efficiency and clients perceive short latencies on unicast. Although the multicast pull channel presents a longer average latency, requests served by it are much fewer than by the other channels. To facilitate later discussion, we define an MPush (or MPull) request as one request that acquires its requested data from the multicast push (or multicast pull) channel.

Our next analysis looks at the latency distribution. We count the requests with latencies in the range from $50i$ to $50(i+1)$ milliseconds, $i \geq 0$. Fig. 3 shows the results with the horizontal dimension denotes the latency. To be more readable, each figure is separated into two parts by a vertical solid line: the left part expresses latencies in a linear scale whereas the right part in a logarithmic form. As seen in Fig. 3(a), 75.2% of all requests experience latencies less than 4 seconds and 96.6% of all requests have latencies within 8 seconds. From the perspective of individual channels, an value of 8 seconds bounds the latency of 95.7% of MPush requests, 89% of MPull requests, and 99.7% of requests served by unicast.

5.2 Peak Time Performance

The effectiveness of hybrid data delivery largely relies on its ability to cope with peak time workloads. Fig. 2(d) exposes the busiest traffic occurred near 8:55pm. For our analysis, we choose an 11-minute peak time that starts from 8:49:00pm to 8:59:59pm. The total requests generated during this period account for 13.5% requests of the entire investigated interval, and create a consistent workload above 36000 requests per minute. Among these requests, we find 799 (0.2%) timeouts. Apparently, the peak workload does not raise the pecentage of timeouts. Table 3 presents the summarized result for comparison with that in Table 2. Our first observation is that the fraction of requests served by multicast push jumps from 74.7% in Table 2 to 86.6% while their average latency rises by 30 percent. As expected, the peak workload forces the MBDD server to select more hot documents. Consequently, the expanded volume of hot data extend multicast cycles, which in turn results in longer latencies perceived by clients. Meanwhile, the unicast channel sees a 50 percent increment in average latency. However, heavy workloads have limited impact on the unicast performance because document selection limits the request serving rate on the unicast channel. An interesting finding is that average latency on the multicast pull channel drops by 25 percent of the amount in Table 2. Multicast pull performance is affected more by workload dynamics than by workload volumes.

5.3 Server Perceived Workload Control

We examine the investigated interval in a granularity of minute. For simplicity, we number the minutes from the beginning of the interval from 0 to 135, and group all requests based on the minute within which they are created. Fig. 4(a) plots the overall request rate issued by clients as well as the request rates satisfied through each channel in every minute. The server successfully controls its perceived unicast workload. When the unicast request rate is much smaller than 100 req/sec, the unicast bandwidth is the dominant bottleneck in the corresponding hot document selection. On the other hand, when the overall workload grows, MPush requests increase steadily and the increment curve is roughly parallel to that of the overall request rate. Fig. 4(a) also shows that the server frequently resorts to multicast pull to avoid overloads when bursty unicast requests arrive. For example, among the requests created in the 94^{th} minute, 2995 of them are satisfied by the multicast pull channel, corresponding to 39% of total unicast requests generated in that minute. However, the benefit of multicast pull is less pronounced during the workload rapidly decreasing phase from the 120^{th} to 128^{th} minute. In this period, hot document selection often over-estimates the future traffic, which results in sufficient available unicast bandwidth.

Fig. 4(b) shows the performance perceived on each channel. The average latency on multicast push climbs

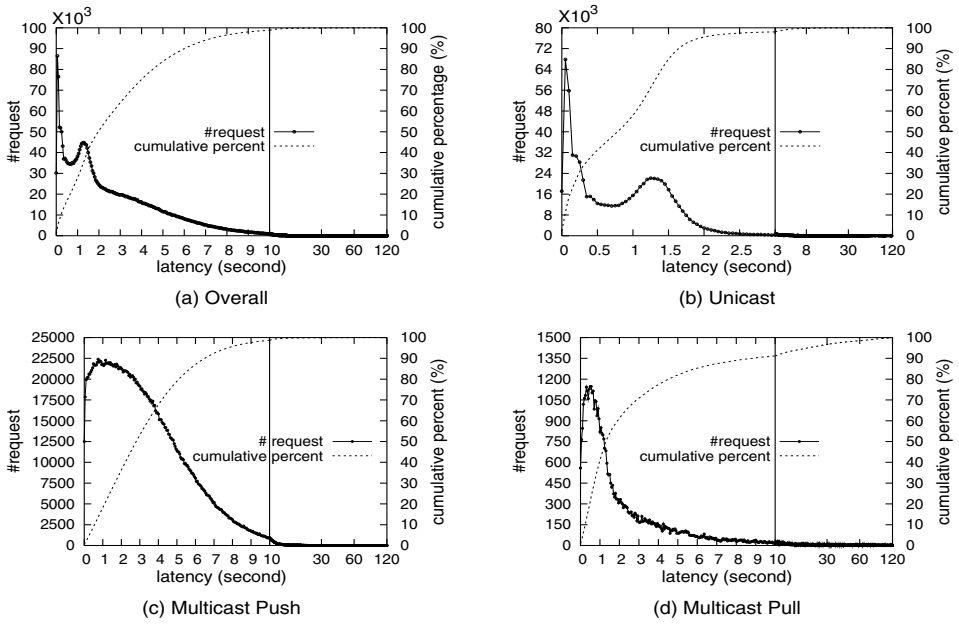

Figure 3. Latency Distribution

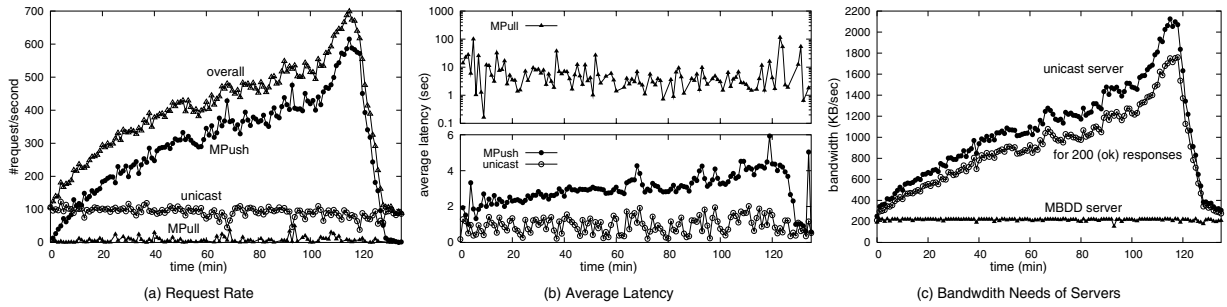

Figure 4. Server Perceived Workload Control

due to a growing amount of hot data caused by the ascending load. However, the increment of latency is slow compared with that of the overall requests shown in Fig. 4(a). As additional requests on hot data will not add a burden on the server, multicast push effectively alleviates the impact of the mounting overall workload. On the "MPush" curve in Fig. 4(b), sharp spikes like those at the 4^{th} and 134^{th} minute occur randomly and in DS periods when the MPush requests are few. However, spikes like that at the 119^{th} minute emerge constantly in our experiments and we give schemes to smoothen them in Sec. 5.5.

Fig. 4(b) also indicates the mean latency on unicast is not notably affected by the variation of workloads. Thanks to hot document selection and multicast pull, the server assures the unicast workload is within the capacity of the unicast channel.

The average latency on the multicast pull channel varies sharply at different times. Despite the acute spikes seen from the figure, we find that there are few requests in the minute where a spike appears. This result is consistent with our previous observation that the latency of most requests is bounded within a reasonable extent.

5.4 Bandwidth Usage

Fig. 4(c) contrasts the bandwidth needed by the MBDD server and a web server to satisfy the requests in the experimental trace. The necessary bandwidth of a web server increases consistently with the growth of workloads. At the busiest moment, a web server requires about 9 times of bandwidth utilized by the MBDD server. In practice, the HTTP protocol allows web servers to send 304 (not modified) responses to reduce data transfer if the requested content is cached at clients

and unchanged. Leaving out requests that cause 304 responses, the curve "for 200 (ok) responses" in Fig. 4(c) depicts the bandwidth used by a web server to satisfy the remainder requests in the experimental trace.

Note that bandwidth saving is only one facet of the benefit of hybrid data dissemination. As the rate of unicast requests is controlled, the server's processing time is also greatly saved.

5.5 Unpopular Large Documents (ULDs)

ULDs can produce remarkable negative impact on performance: they cause sudden increase of multicast push latency. Examples can be seen at the 119^{th} minute in Fig. 4(b). A ULD has two features. First, its size is notably large compared with the average document size. Second, its popularity is extremely low and requests on it appear sparsely, i.e., it is always cold.

Suppose a ULD D with size s is accessed once in the previous DS period. In hot document selection at the beginning of the current DS period, the selection algorithm (Sec. 3.3) will assume the request rate on D to be $1/L$ and allocate s/L of unicast bandwidth for D, where L is the DS period length. Since s is huge, the allocation takes away much unicast bandwidth. As a result, the algorithm has to select more hot documents. The boosted volume of hot data causes a sudden increment of latency perceived on the multicast push channel. For instance, at the 118^{th} minute, the server receives 3 requests on 3 different ULDs with a size of 2.1MB, 1.9MB and 1.4MB respectively. In the 119^{th} DS period, the total size of hot data increases by 79 percent of that of the previous period. Consequently, the mean latency of multicast push at the 119^{th} minute increases by 32 percent (see Fig. 4(b)).

The effect of ULDs is largely due to fact that the selection algorithm overestimates the popularity of ULDs. The algorithm can only distinguish a minimum request rate of $1/L$ while the actual request rate on a ULD can be far below it. A natural solution is to employ enlarged DS periods particularly for ULDs. However, using various lengths of DS periods introduces complexity in the server. We have devised a simple and efficient solution.

Essentially, our solution breaks a ULD into small sections and transfers them across multiple DS periods. We limit the unicast bandwidth allocated to a ULD in each DS period so that ULDs would not consume excessive bandwidth, which avoids sudden increase of hot data size. Since the ULD transmission is broken down across multiple DS, its transmission time is lengthened appropriately depending on the document size and the server's load. Suppose a ULD's size is no less than S. Two modifications are made to the hot document selection algorithm. (1) If a ULD D with size s is accessed n times in the previous DS period, the algorithm assumes D has $\lceil s/S \rceil$ sections numbered from 0 to $\lceil s/S \rceil - 1$. Each of the first $\lceil s/S \rceil - 1$ section has a size of S. The bandwidth needs of a section are an/L, where a is the section's size. The algorithm uses the 0^{th} section for the current document selection and allocates Sn/L unicast bandwidth to the section, that is, D is treated as if its size were S. The algorithm adds the future bandwidth need of the i^{th} section ($i > 0$) to the $(i-1)^{st}$ slot in a history Q that represents the cumulative future bandwidth needs of the ULDs. (2) At the beginning of the document selection algorithm, the head of Q is removed and its value is interpreted as the cumulative unicast bandwidth needs of ULDs inherited from previous DS periods. The algorithm subtracts this inherited bandwidth needs from the unicast bandwidth parameter U before it selects new hot documents.

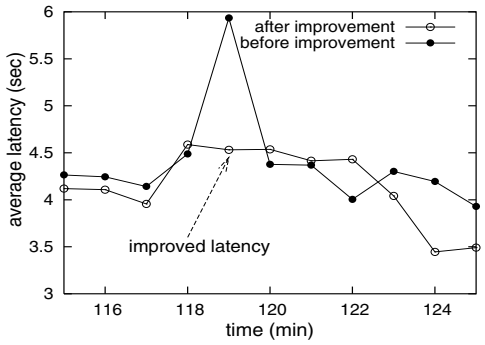

Figure 5. Improved Multicast Push Performance

In Fig. 5, we give the experiment results with the improved selection algorithm. We specify $S = 500$KB. The latency on multicast push at the 119^{th} minute declines for the hot data size is effectively reduced using the improved algorithm. Moreover, the figure shows the inherited bandwidth needs do no notably affect the multicast push performance after the 119^{th} minute. In our experiment, we also observe that the latency on unicast and multicast pull do not change appreciably (not shown in the figure). As a disadvantage, the mean latency of the 3 requests for the 3 ULDs at the 118^{th} minute is roughly doubled from 68 to 145 seconds. However, users generally will tolerate more latency in retrieving a larger document. Also, there are not many users waiting for a ULD.

6 Synthetic Experiments

We have reported the details of our synthetic experiments in [31]. Due to space constraints, we only

highlight the results here. In the first experiment, we stress the adaptation speed of the server. The synthetic workload increases from 0 to 1000 req/sec instantly and maintains 1000 req/sec thereafter, which simulates the rapid and large workload changes such as flash crowds [21]. We observe that our server monitors the workload effectively under extreme heavy traffic, and adapts to the workload change by a single hot document selection. Our second experiment scenario demonstrates the effectiveness of our warm document selection scheme from the perspective of the client perceived latency. We use the same synthetic workload and perform the experiment with warm document selection disabled. We observe that the average client perceived latency is increased by a factor of 3.5.

7 Related Work

Multicast push and multicast pull have been previously combined in a hybrid scheme using static selection of hot documents [2]. Hybrid models with multicast pull and unicast have also been analyzed, and the delivery method for a document is determined by the integrated schedule for both channels [4]. Other work examines hybrid schemes using multicast push and unicast [7, 11, 23, 28]. Some of them study document selection in combination with the problem of bandwidth division among the channels [7, 11]. They assume flat multicast and arbitrary bandwidth can be used for multicast. The multicast bandwidth has also been modeled as constant like in our work [23, 28]. A document selection approach has been proposed for servers using flat multicast [28]. In another method, the server monitors its load and increases (or decreases) the amount of hot data when the load is high (or low) [23]. The adjustment can occur at any time as needed. A comprehensive review of hybrid schemes is in our survey [14].

The popularity of a hot document can be estimated by *probing* [28]. The server temporarily removes the document from the push channel, and the number of solicited requests indicates its popularity. This method might suffer from scalability problem if the document is extremely popular [11]. Alternatively, access information of hot data is piggybacked to unicast requests [23]. However, accurate popularity of hot data cannot be obtained since clients do not report references to hot data if they do not access cold documents.

In our work, hot document selection is periodically performed. The selection algorithm does not require a specific multicast schedule. Popularity of hot data is collected by sampling clients. Unlike prior work, our server combines three channels. The difference is not trivial. For example, in our model, multicast pull should be conservative and is used to relieve sudden and heavy load; but in a model using multicast push and multicast pull, the role of multicast pull is different as it serves all unicast requests, and multicast pull performance has to be investigated more carefully. Multicast pull also makes our server more robust to workload dynamics than hybrid servers that only use multicast push and unicast.

Collecting cold data popularity is seldom studied in earlier research. The issue is important for server to survive unexpected environment changes, such as flash crowd [21], a sudden large surge of traffic that often results in server collapse. In our design, when flash crowd occurs, despite requests are frequently dropped by the server at the beginning, the unicast workload is accurately monitored. Consequently, hot document selection is effectively performed, thereby leading the server to adapt promptly.

8 Conclusion

We have studied the key issues that arise in document selection in hybrid data delivery over the Internet. While previous related work often focuses on one or several separate issues, we attempt to devise integrated solutions for the document selection problem. Moreover, prior related work rarely provides quantitatively evaluation in the real Internet with real server traces. Our proposed solutions are implemented and evaluated with real web traces and on PlanetLab, a real Internet based test-bed environment. Our results show that our multicast server can effectively adapt to dynamic environments and is substantially more scalable than traditional Web servers. Our work is a contribution to building practical hybrid data dissemination services.

Acknowledgements

We would like to thank PlanetLab for providing the network testbed and the anonymous IPDPS 2008 reviewers for their helpful comments.

References

[1] PlanetLab. *http://www.planet-lab.org/*.

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

[3] S. Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. *MobiCom*, 1998.

[4] M. Agrawal, A. Manjhi, N. Bansal, and S. Seshan. Improving web performance in broadcast-unicast networks. *In Infocom*, 2003.

[5] S. Alouf, E. Altman, and P. Nain. Optimal on-line estimation of the size of a dynamic multicast group. *In Infocom*, 2002.

[6] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *Tech. Report, HP Lab*, 1999.

[7] Y. Azar, M. Feder, E. Bubetzky, D. Rajwan, and N. Shulman. The multicast bandwidth advantage in serving a web site. *In NGC*, 2001.

[8] F. Baccelli, A. Chaintreau, Z. Liu, and A. Riabov. One-to-many TCP overlay: A scalable and reliable multicast architecture. *Infocom*, 2005.

[9] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *In ACM SIGCOMM*, 2002.

[10] A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber. Minimizing service and operation costs of periodic scheduling. *Math. Oper. Res.*, 27:518–544, 2002.

[11] J. Beaver, N. Morsillo, K. Pruhs, P. Chrysanthis, and V. Liberatore. Scalable dissemination: What's hot and what's not. *In WebDB*, 2004.

[12] J. Beaver, K. Pruhs, P. Chrysanthis, and V. Liberatore. The multicast pull advantage in dissemination-based data delivery. *Proceedings of 3rd Hellenic Data Management Symposium*, 2004.

[13] P. Chrysanthis, V. Liberatore, and K. Pruhs. Middleware support for multicast-based data dissemination: A working reality. *WORDS*, 2003.

[14] P. K. Chrysanthis, V. Liberatore, and K. Pruhs. Middleware for scalable data dissemination. In Q. H. Mahmoud, editor, *Middleware for Communications*, pages 236–260. Wiley, 2004.

[15] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. *In Proc. of ACM SIGMETRICS*, pages 1–12, 2000.

[16] B. Chun and A. Vahdat. Workload and failure characterization on a large-scale federated testbed. *Tech. Rep. IRB-TR-03-040, Intel Research Berkeley*, 2003.

[17] P. Francis. Yoid: Extending the internet multicast architecture. *Technical report, AT&T Center for Internet Research at ICSI (ACIRI)*, 2000.

[18] H. Frystyk-Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Wium-Lie, and C. Lilley. Network performance effects of http/1.1, css1 and png. *In ACM SIGCOMM*, 1997.

[19] Y. Guo, M. Pinotti, and S. Das. A new hybrid scheduling algorithm for asymmetric communication systems. *ACM SIGMobile Computing and Communications Review*, 5(3):123–130, 2001.

[20] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. W. O'Toole. Overcast: Reliable multicasting with an overlay network. *In Proc. of OSDI*, 2000.

[21] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. *In WWW*, 2002.

[22] J. Liebeherr, M. Nahas, and W. Si. Application-layer multicast with delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, 20(8):1472–1488, 2002.

[23] C. Lin, H. Hu, and D. Lee. Adaptive realtime bandwidth allocation for wireless data delivery. *Wireless Networks*, 10(2), 2004.

[24] R. Rivest. The MD5 message-digest algorithm. *RFC1321*.

[25] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology*, 6(1):20–52, 2006.

[26] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. *In USENIX FAST*, 2005.

[27] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application endpoints. *In SIGCOMM*, 2004.

[28] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. *In VLDB*, 1997.

[29] C. Su, L. Tassiulas, and V. Tsotras. Broadcast scheduling for information distribution. *In INFOCOM 1997*, 1997.

[30] J. Xu, X. Tang, and W. Lee. On scheduling time-critical on-demand broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):3–14, 2006.

[31] W. Zhang. Scalable hybrid data dissemination for internet hot spots. *PHD's Thesis, Case Western Reserve University, Cleveland, Ohio, USA*, 2007.