

# Pull vs Push: A Quantitative Comparison for Data Broadcast

Demet Aksoy    Mason Sin-Fai Leung  
Computer Science Department  
University of California, Davis  
(aksoy,leungm)@cs.ucdavis.edu

**Abstract**—Advances in wireless technology have resulted in an emerging broadcast-capable infrastructure. A major debate for such infrastructures is whether we should use *push* or *pull* to support large client populations. To date, push was suggested to provide a scalability that is not possible with pull. In this paper we conclude otherwise based on simulation and experimental prototype results. A major contribution of this paper is to show that pull-based algorithms outperform push-based ones. Contrary to the conventional wisdom, this is true even for environments where request processing and scheduling overheads are a significant factor.

## I. INTRODUCTION

Due to the explosive growth in the number of users, and the amount of on-line data, *scalability* concerns are more pressing than ever. It has been long argued that traditional unicast delivery can not scale for such drastic workloads. A significant portion of emerging Internet infrastructures are already broadcast capable, i.e., satellite networks, wireless networks, CATV, Ultra WideBand technology, etc.

A major question is “*how such infrastructures should employ broadcast for large-scale applications*” like emergencies, election and sporting event result servers, news and entertainment delivery, etc. In general, data communication systems can be distinguished according to whether they are based on a *push* model or a *pull* model. Using push, data items are sent out to the clients without explicit requests. In contrast, with pull data items are broadcast in response to explicit requests received from clients. Pull can be used either for unicast or for broadcast. When used for broadcast, pull is also referred to as *interactive*, or **on-demand** broadcast.

It has been long argued that push is more beneficial than pull since it provides tremendous scalability, i.e., increasing the number of users does not increase the average responsiveness of the system. A few examples include:

- Broadcast Disks [1][2] project showed that push performance was better than pull performance.
- Hu and Lee [3] showed that pull-based performance is not good as push-based performance except for light loads.
- In 2001 Deolasee et al. [4] proposed a hybrid push-pull to improve performance.
- In 2002 Hu and Chen [5] showed that push is better than pull for heavy loads.
- Stathatos et al. [6] suggested a hybrid pull/push.

When considering these studies it should be noted that the majority [5] [4] [3] [6] have used pull as a unicast delivery

method. In other words, transmission of a data item was repeated for *each* request even though the medium was based on broadcast. The comparison in such a case is dominated by the difference between unicast and broadcast, rather than pull and push. In studies that considered broadcast based pull, on the other hand, the broadcast scheduling was based on FCFS which has a poor performance.

To the best of our knowledge, this paper is the first study towards a comprehensive comparison of push and pull for broadcast systems. In this study we provide a detailed performance evaluation based on a broadcast testbed in addition to simulations.

A major contribution of this paper is to show that, despite the common wisdom, pull-based data broadcast outperforms push even for settings where request processing and scheduling overheads result in significant performance deterioration. Our results suggest that this is true even for extremely high request rates. This has important implications for data center designs, and highly scalable wireless communications.

In the following we first present our performance criteria. In section III, we describe the pull and push broadcast algorithms that we employ in this study. We then describe our **simulation** results followed by our **experimental** results derived from our data broadcast testbed in section IV and conclude with section V.

## II. PERFORMANCE CRITERIA

Our comparison focuses around the following metrics:

**Average Waiting Time:** As our primary metric, we measure the amount of time on average, from the instant that a client request arrives at the server to the time that the requested item is broadcast.

**Worst Case Waiting Time:** We also measure the maximum amount of time that any *individual* client waits for service to account for fairness.

**Efficiency:** Push broadcast models assume some *a-priori knowledge* of client access distribution and prepare an *off-line* schedule. With a pull-based model, on the other hand, *online* scheduling is required, since, the server must choose which item to broadcast among those that have outstanding request(s) at each instance. In this respect we measure the data broadcast rate that can be obtained to reflect the overhead.

**Scalability:** As the amount of data available on-line is increasing in addition to the number of users, it is important to scale for large database sizes, as well as users.

**Robustness:** Due to the dynamic nature of demand, it is important to sustain a good performance regardless of a specific access pattern.

### III. BROADCAST ALGORITHMS

In this paper, we study a wide range of broadcast algorithms in two major categories: pull and push.

#### A. Pull Algorithms

In a broadcast pull environment, responses to requests are transmitted on a dedicated shared broadcast channel. Following are the specific algorithms that we study:

**First Come First Served (PULL FCFS):** [7] broadcasts the data items in the order they are requested. To avoid redundant broadcasts, a request for a data item that is already in the queue is ignored.

**Longest Wait First (PULL LWF):** [8] schedules the data item that has the largest total waiting time, i.e., the sum of the time each pending request for the item have been waiting for. We implement LWF using three values for each unique item requested: 1) the number of outstanding request(s) for the data item ( $R$ ), 2) the last recording of the total waiting time, and 3) the time of this recording. For instance, at time  $0$ , when a data item is requested for the first time, the three fields of its service queue entry are set to  $(1, 0, 0)$ . A subsequent request at time  $t$  results in an update as  $(2, 0+1 \times (t-0), t)$ . We repeat the update process for every new coming request and also at the time of scheduling decision to avoid using stale total waiting time values.

**PULL RxW:** [9][10] selects the data item with the maximal  $R \times W$  value where  $R$  is the number of outstanding requests for that item and  $W$  is the amount of time that the oldest outstanding request for that item has spent in the service queue. Using two threaded lists that keep the service queue entries sorted in  $R$  and  $W$  order, the scheduling decision time is significantly reduced.

**PULL RxW. $\alpha$ :** [9][10] An approximate version of the algorithm reduces the scheduling decision time even further. The algorithm broadcasts the *first* item it encounters whose  $RxW$  value is greater than or equal to  $\alpha$  times the current *threshold* value. The *threshold* is computed as the running average of the  $RxW$  value of the last item broadcast and the previous *threshold*. After each broadcast decision, the *threshold* is updated accordingly.

The scheduling overhead decreases as alpha decrease. In the experiments we use two settings for the approximation algorithm: **PULL RxW.90**, ( $\alpha = 0.9$ ), and **PULL RxW.0**, ( $\alpha = 0$ ).  $RxW.0$  compares only two items, the item with the most outstanding requests and the one that has been waited for the longest time.  $RxW.0$  is a constant-time algorithm.

#### B. Push Algorithms

For our performance evaluation, we use the following push schedules:

**PUSH BDisk (Broadcast Disks):** [1] In this schedule all items being accessed are partitioned into a number of

disks. Items on the smaller numbered disks are broadcast more frequently than those on the larger numbered disks. For instance, for 2 disks with items  $\{a\}$ , and  $\{b, c\}$  and relative frequencies of 2 and 1, the schedule should look like  $a, b, a, c, a, b, \dots$ . We use the *optimized* parameters for Broadcast Disks [11] assuming perfect knowledge of access probabilities.

**PUSH MAD:** [12] selects the data item whose *access probability  $\times$  time since last broadcast* is the highest. In the extreme case, MAD requires examining all items in the database before making a scheduling decision. In the experiments we run MAD twice; in the first run we record the schedule generated, and in the second run we use this recorded schedule when taking our measurements. This results in an optimistic off-line algorithm with the highest throughput.

**PUSH LB (Push Lower Bound)**<sup>1</sup>: [7][13] The optimum schedule for push is defined by Ammar et al. [7] as follows: 1) popular items should be broadcast more often than their not-so-popular counterparts, 2) two successive broadcasts of the same item should be equal distance throughout the schedule, and 3) the relative number of appearances of items on the broadcast channel should be in the square root ratio of their access probabilities. In other words, two items that have access probabilities  $p_i$  and  $p_j$  should be broadcast with the ratio  $\frac{\sqrt{p_i}}{\sqrt{p_j}}$ . For instance, if the access probability of item  $i$  is 0.4, and of page  $j$  is 0.1, then item  $i$  should be broadcast twice as often as item  $j$ . The theoretical lower bound on average waiting time is equal to  $\frac{1}{2}(\sum_{i=1}^N \sqrt{p_i})^2$ , where  $N$  is the size of the database, and  $p_i$  is the access probability for item  $i$ . In order to calculate the theoretical lower bound we assume that we can use the full bandwidth and an ideal allocation is possible. In practice, it is *not* always possible to generate such a schedule due to timing conflicts, i.e., two pages need to be broadcast at the same time.

It should be noted that we assume **perfect knowledge** of access probabilities when determining push schedules.

### IV. PERFORMANCE EVALUATION

In this section we present our simulation and prototype based performance evaluation results. In this study we use fixed-length data items, referred to as *pages*. In our default setting, we use a database size of 10,000 pages accessed with a Zipf distribution which is shown to be a realistic approximation of skewed data access [14]. Zipf distribution is such that the probability of accessing page  $i$  is  $p_i = \frac{1}{i^\theta \sum_{j=1}^N \frac{1}{j^\theta}}$ , where  $N$  is the size of the database, and  $\theta$  is the skewness parameter. As  $\theta$  value is increased the skewness increases. In the experiments we use ( $\theta=1$ ) as a default value and then vary this parameter to study sensitivity to different access patterns. We use a batch workload generator [15] running on a dedicated machine that produces item requests according to the access distribution, and sends them to the server over the uplink.

For the experiments, we run the system until equilibrium is reached, i.e., when the request satisfaction rate converges to the request arrival rate, before taking measurements. During the experiments, the testbed is isolated from the external network to avoid external network traffic bias the measurements.

<sup>1</sup>PUSH LB represents the analytical lower bound that is usually *not possible* to achieve for push-based algorithms. It is not an implemented algorithm.

## A. Simulation Results

In this section we provide the results of our simulation study for a basic comparison between the lower bound of push algorithms (PUSH LB) and various pull algorithms. In our simulation study, we use a logical time unit of *ticks* as in previous work. *Tick* refers to the time required to broadcast a single page – with the assumption that the time to broadcast a page is constant across all algorithms. Note that the overheads of making scheduling decisions or processing requests are neglected in the simulation studies.

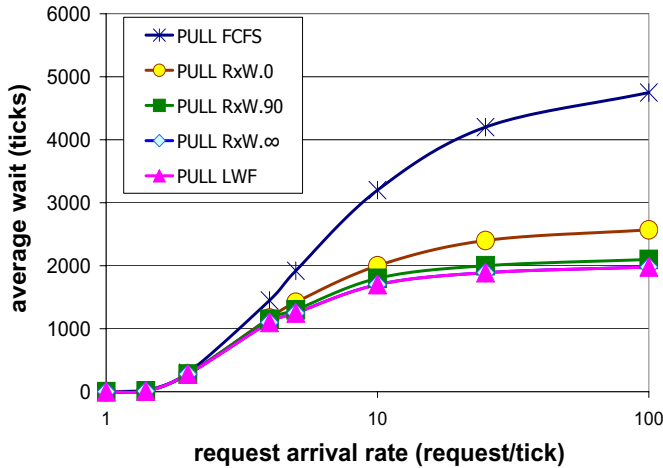


Fig. 1. **Simulation Results:** FCFS provides the worst average waiting time among all PULL algorithms.

In figure 1 we plot the average waiting time as request arrival rate increases. We observe that the average waiting time of all pull-based algorithms increases as request rate increases. This increase, however slows down for high request rates, resulting in a flattening out of curves. After such flattening out takes place we observe a performance order from the worst to the best as: FCFS, RxW.0, RxW.90, followed by LWF and RxW. $\infty$  (the two overlapping curves). The lower bound of push, PUSH LB, is slightly over 2000 ticks in this setting.

## B. Experimental Results

To evaluate the impact of overheads ignored in simulation studies, we have implemented a data broadcast server testbed on a cluster of Pentium-based machines running Windows NT 4.0. Each machine in the cluster has two independent network connections for a 1) 10Mbps uplink and a 2) 100Mbps downlink. Over the uplink clients send requests to the server using TCP/IP. The server broadcasts the requested items to clients over the downlink using UDP. The server used for these experiments has a Pentium III 800MHz processor and 128 MB memory. We now turn to the evaluation of pull and push for data broadcast using our experimental test-bed.

1) **Average Responsiveness and Efficiency:** In the first set of experiments using the prototype, we measure the average waiting time for the same setting as in the simulation study.

Please note that unlike the simulation study, however, the unit of time is now the actual time in *seconds* rather than *ticks*. The default page size is 16Kbytes (varied later). The optimized BDisk parameters [11] suggest three disks with page partitions of [1..102], [103..1555], [1556..10000] and relative frequencies of (15, 3, 1).

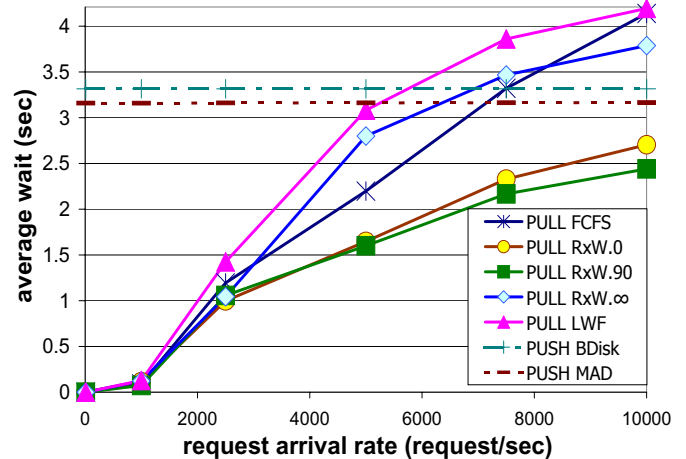


Fig. 2. **Prototype Results:** throughout the whole range PULL (RxW.90 and RxW.0) has the best performance in terms of average waiting time.

Figure 2 plots the average waiting time as request arrival rate increases. Pull based algorithms (FCFS, LWF, RxW. $\infty$ , RxW.90 and RxW.0) are plotted using solid lines, and push-based algorithms (PUSH MAD and PUSH BDisk) are plotted using dashed lines. When we compare push and pull at 1 req/sec (3600 requests per hour) pull algorithms drastically outperform push algorithms, since pull algorithms reply requests as soon as they arrive. The gap between push and pull closes for higher request rates. However, throughout the whole range, pull-based algorithms (RxW.90 and RxW.0) have the best performance.

As can be seen in figure 2 the average waiting time of push based algorithms is independent of the request arrival rate. We also computed the theoretical lower bound PUSH LB in this setting assuming the maximum achievable data rate and ideal scheduling as 2.88 seconds – still worse than PULL RxW.90 and RxW.0. The average waiting time of pull-based algorithms increases as request rate increases, and the curves eventually flatten out for high request rates [9]. Unlike the simulation results however, we observe a new performance order among the algorithms. For example, FCFS, which was suggested to be the worst pull algorithm in simulations, now performs better than LWF. This difference is due to the efficiency of FCFS compared to LWF. Our LWF implementation performs an exhaustive search on the service queue. Whereas FCFS maintains an extremely simple service queue with constant processing time: Similarly, low overhead RxW.90 and RxW.0 perform significantly better than high overhead ones.

In order to better compare the experimental results with the simulation results we plot the average number of requests received between two successive broadcasts (requests per tick

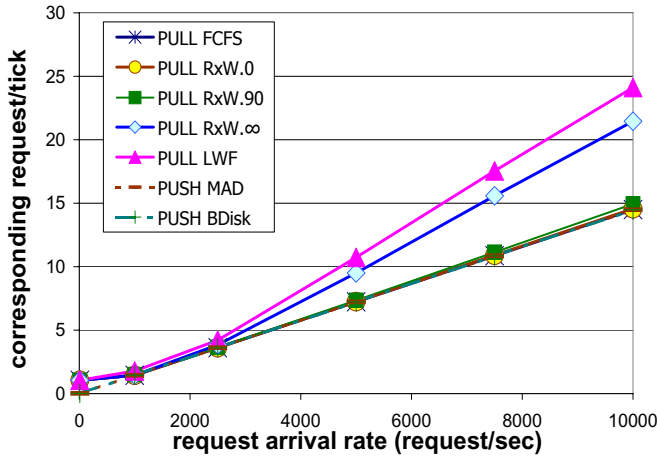


Fig. 3. Request Rate Conversion: A request rate of **36 Million requests** per hour (10,000 requests/sec) corresponds to only 15 requests per tick for most algorithms.

as in simulation studies) for the same experiment. At 10,000 req/sec we see that RxW.90 receives about 15 requests between two successive broadcasts. This means that we should be looking at RxW.90's position at 15 requests/tick in the simulation results when analyzing this point. Figure 3 can also be used to extrapolate the point where RxW.90 will have similar performance to implementable PUSH algorithms. Using a conservative approach where we assume that overheads will have a greater impact as request rate increases, we estimated this point to be around 65,000 requests/sec. Note that this number is more than three-fold compared to the *peak* request rate for *cnn.com* [16]. Thus we conclude that pull broadcast scales even for extremely heavy loads.

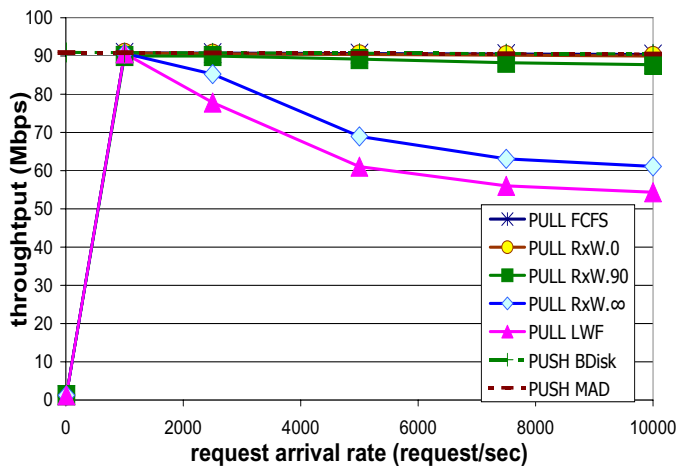


Fig. 4. Low overhead PULL algorithms are as efficient as PUSH algorithms.

The effective broadcast throughput based on transmitted data for this experiment is plotted in figure 4. In our experimental setting the maximum achievable throughput is measured to be 91Mbps. All push algorithms achieve the

maximum throughput since they use off-line schedules. For pull algorithms, at 1 request/sec, the server is mostly idle since requests are replied as soon as they arrive and the service queue is emptied immediately. As the request rate increases to 1000 request/sec, such idle time diminishes and pull-based algorithms start using almost all of the available bandwidth. This peak utilization is sustained by efficient pull algorithms. FCFS, RxW.0, and RxW.90 stay very close to PUSH algorithms.<sup>2</sup> In the figure they appear as overlapping lines at this scale (only RxW.90 is visibly lower). The bandwidth utilization of LWF and RxW.∞ drops significantly beyond 1000 request/sec. This behavior is due to two factors that depend on each other. 1) LWF and RxW.∞ are high overhead algorithms; they scan a significant number of service queue entries before making a scheduling decision. This increases the amount of bandwidth that goes unused while the server is busy with these overheads. 2) As the algorithm spends more time to make the scheduling decision, the number of requests that arrive between successive broadcasts also increases further adding up to the processing overhead.

Our experiment results reveal two important conclusions: (1) the overhead is a significant factor in this experimental setting; (2) despite this factor, efficient and high quality pull algorithms perform better than push algorithms even for request rates that reach up to 10,000 requests/sec (**36 Million requests/hour**).

2) **Individual Responsiveness - Maximum Wait:** In this section we shift our focus from the traditional metric of *average waiting time* to the *maximum waiting time* that is experienced by *any* client. Figure 5 plots the maximum waiting time for the same experiment focusing on efficient pull algorithms. Pull algorithms broadcast pages only when they are requested. Therefore, for low request rates the worst case behavior of pull is orders of magnitude less than that of push.

RxW.0 and FCFS provide the best performance within the whole range, and RxW.90 is significantly better than PUSH MAD. Recall that BDisk results in the worst push performance in terms of average wait.

As expected, for push-based algorithms the worst case waiting time depends on the total length of the schedule that is prepared; i.e., the time between two successive broadcasts of the least frequently broadcast page. For 10,000 pages accessed by a Zipf distribution ( $\theta = 1$ ) PUSH BDisk needs to broadcast 14,334 pages before every page appears on the broadcast channel at least once.

3) **Scalability:** In this section we study the performance of push and pull for varying the number of pages. We observe that the average waiting time of PUSH increases as database size increases due to the relative increase in the number of pages that need to be broadcast. In fact, if we consider the amount of the on-line data within the Internet, the broadcast cycle of push schedules would converge to infinity. For pull-

<sup>2</sup>Recall that FCFS and RxW.0 are constant time algorithms.

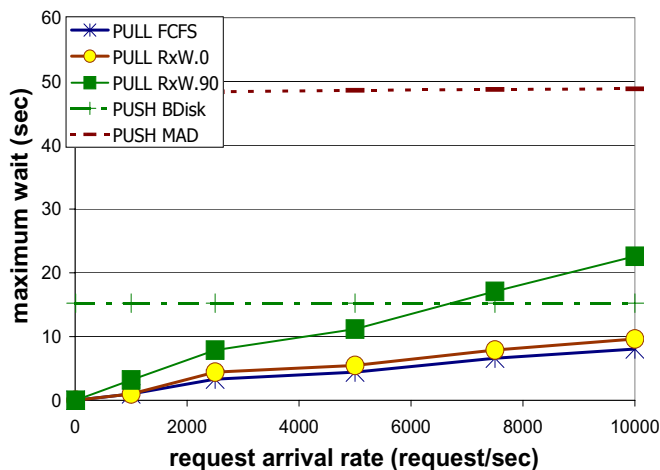


Fig. 5. PULL FCFS, RxW.0 and RxW.90 receive the least popular items earlier than PUSH MAD.

based algorithms, however, the increase is almost negligible, since pull-based algorithms send *only* requested pages.

The maximum waiting time of algorithms is also high for PUSH for larger database sizes due to excessive amount of broadcasts to complete the period. Our analysis suggest that pull scales much better than push in the face of increasing database sizes.

4) **Robustness:** We complete our analysis with a sensitivity analysis of the results that we have obtained in the previous experiments. In the first experiment we evaluate the performance for varying item sizes to study the effects of different broadcast rates. For instance, small page sizes correspond to higher broadcast bandwidths such that the time to broadcast a single page decreases. For the first set of experiments, we observe that the average waiting time of push algorithms increases linearly as the page size increases. This is due to the fact that the push schedules are static for a given access probability and as the page size increase the time to broadcast a page increase. In case of pull, for page sizes upto 16K we do not observe a significant increase in average waiting time. As page size increase overhead ceases to be an issue, and the performance order of the algorithms converge to that suggested by the simulation study. Within the whole range that we have tested, we observe that RxW.90 and RxW. $\infty$  provide the best performance in terms of average waiting time. RxW.0 and FCFS have the lowest maximum waiting time in the whole range. Our conclusion is that RxW.90 provides the best trade-off between individual and overall performance in this setting.

For systems that have a faster CPU we expect to see the results for pull to be even better than those that we have presented, as overhead becomes less of an issue.

We have also performed experiments to evaluate the algorithms for different access distributions. We varied the access skew parameter of Zipf,  $\theta$ . When  $\theta$  is set to 0 the access distribution is uniform. For various  $\theta$  values we have updated the schedule for PUSH BDisk according to the optimum numbers based on parameters provided by the authors of [11]

assuming perfect knowledge of access probabilities. In our experiments we observe that as skewness decrease, i.e.,  $\theta$  value is decreased, all algorithms relatively deteriorate. Within the whole range we studied we observed RxW.90 to provide the best performance.

## V. CONCLUSIONS

Previous studies suggest that push has a better performance than pull. These studies have either used pull as a unicast approach even though the communication medium was based on broadcast, or they have used a poor scheduling algorithm to represent pull. In this paper we have provided a quantitative comparison of push-based and pull-based data broadcast using simulation, and experimental analysis. We first described various pull-based (FCFS, LWF, RxW. $\infty$ , RxW.90 and RxW.0) and push-based (Broadcast Disk, MAD) algorithms. We then described the theoretical average waiting time lower bound for push (PUSH LB) and discussed why such bounds are infeasible for practical systems. We evaluated push and pull using an experimental prototype testbed so that overheads are taken into account in the performance results. Despite the conventional wisdom, our results suggest that, pull-based broadcast outperforms push-based broadcast even for cases where server overhead is a significant factor. Our results suggest that this is true even for extremely high request rates.

## REFERENCES

- [1] S. Acharya, R. Alonso, and M. Franklin, "Broadcast disks: Data management for asymmetric communication environments," in *Proc. ACM SIGMOD*, Santa Cruz, CA, May 1995.
- [2] S. Acharya, M. Franklin, and S. Zdonik, "Balancing push and pull for data broadcast," in *Proc. ACM SIGMOD*, May 1997.
- [3] Q. Hu, W. Lee, and D. Lee, "Performance evaluation of a wireless hierarchical data dissemination system," in *Proc. MOBICOM*, Seattle, August 1999.
- [4] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push-pull of dynamic web data: Better resiliency, scalability and coherency," in *Proc. ACM World Wide Web*, May 2001, pp. 265–274.
- [5] C. Hu and M. S. Chen, "Adaptive balanced hybrid data delivery for multi-channel data broadcast," in *Proc. IEEE International Conference on Communications (ICC)*, Apr 2002, pp. 960–964.
- [6] K. Stathatos, N. Roussopoulos, and J. S. Baras, "Adaptive data broadcast in hybrid networks," *Proc. VLDB*, vol. 30, no. 2, September 1997.
- [7] H. Dykeman, M. Ammar, and J. Wong, "Scheduling algorithms for videotex systems under broadcast delivery," in *Proc. IEEE International Conference on Communications*, Toronto, Canada, 1986, pp. 1847–1851.
- [8] H. Dykeman and J. Wong, "A performance study of broadcast information delivery systems," in *Proc. IEEE INFOCOM*, 1988.
- [9] D. Aksoy and M. Franklin, "RxW: A scheduling approach for large scale on-demand broadcast," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 7, Dec. 1999.
- [10] D. Aksoy, M. Franklin, and S. Zdonik, "Data staging for on-demand broadcast," in *Proc. VLDB (Int. Conf. on Very Large DataBases)*, 2001.
- [11] J. Hwang, S. Cho, and C. Hwang, "Optimized scheduling on broadcast disk," in *Proc. Conf. on Mobile Data Management (MDM)*, 2001.
- [12] C. Su and L. Tassioulas, "Broadcast scheduling for information distribution," in *Proc. IEEE INFOCOM*, 1997.
- [13] M. H. Ammar and J. W. Wong, "On the optimality of cyclic transmissions in teletext systems," *IEEE Transactions on Communications*, vol. 35, no. 1, pp. 68–73, Dec 1987.
- [14] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. IEEE INFOCOM*, 1999.
- [15] W. C. Cheng, C.-F. Chou, L. Golubchik, S. Khuller, and J. Y. Wan, "Large-scale data collection: a coordinated approach," in *Proc. IEEE INFOCOM*, 2003.
- [16] W. LeFebvre, "CNN.com: in LISA 2001," *login: the magazine of USENIX and SAGE*, vol. 27, no. 1, p. 83, Feb. 2002.