Periodic Scheduling in On-Demand Broadcast System

Nitin Prabhu, Vijay Kumar
SCE, Computer Networking
University of Missouri-Kansas City
Kansas City, MO 64110
npp21c (kumarv)@umkc.edu

Indrakshi Ray
Computer Science
Colorado State University
Fort Collins CO 80523-1873
iray@cs.colostate.edu

Abstract: Recent advances in mobile computing have enabled the deployment of broadcast based information systems such as, wireless internet, traffic information systems, etc. These systems are mainly pull-based and their performance very much depends on the broadcast schedule they use. In this paper we focus on on-demand broadcast system. We propose a new on-demand scheduling algorithm that takes scheduling decision at periodic interval, unlike previous algorithms that take decision after broadcasting every data item. This reduces the time client spends monitoring the broadcast channel for data items. We study its behavior with a detailed simulation study and show that our algorithm performs better than the pervious algorithms for on-demand systems.

1 Introduction

Data broadcasting through wireless channels is becoming a common way to reach information seekers to satisfy their demands. Unlike conventional unicast approach, it offers high scalability and is capable of satisfying multiple requests for the same data item at once which obviously leads to efficient bandwidth utilization. One of the main components of broadcasting systems is broadcast scheduling algorithms, which significantly affect data latency and data access time and is the topic of this paper. In order to develop the motivation, we first categorize existing broadcast approaches and identify their limitations.

Broadcast systems can be categorized into (a) push-based, (b) pull-based, and (c) hybrid. In push-based system server periodically broadcasts a schedule, which is computed offline using user access history. This approach is also referred to as *static* broadcast which does not take into account the current data access pattern. Its performance is significantly affected when user access pattern deviates from the one, which was used to construct the broadcast schedule. In pull-based systems, which are commonly referred to as on-demand broadcast system, clients explicitly request for data items from the server. The server compiles the request in service queue and based on number of pending requests for the data items, broadcasts the data. Unlike push-based system, the pull-based systems perform better mainly because they make decision based on current user access pattern. Hybrid systems apply a mixture of push and pull approaches where rarely accessed data items are classified as pull and commonly accessed data items are periodically pushed.

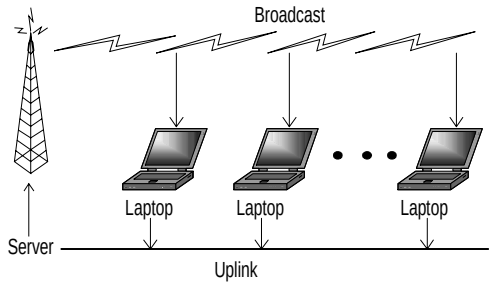


Fig. 1 On-demand Broadcast System

In this paper we focus on *on-demand* broadcast system because (a) it makes efficient use of available bandwidth, (b) it allows timely access to data, and (c) majority of users frequently seek specific information. We propose a new on-demand scheduling algorithm that takes scheduling decision at periodic interval, unlike previous algorithms that take decision after broadcasting every data item. The important implication of this is it gives periodic nature to the on-demand data broadcast and indexing can be used for accessing broadcast data. This reduces the time client spends monitoring the broadcast channel for data items. In the previous works on on-demand algorithm it was not possible to use indexing and clients had to continuously monitor the channel till its desired data item was broadcasted. To establish its usefulness, we study its behavior with a detailed simulation study and show that compared to our algorithms performs better than are pervious algorithms.

The remainder of this paper is structured as follows: Section 2 discusses the related works and motivation for our work; section 3 discusses new issues in transactional requests and motivates the need for new scheduling algorithm. In Section 4 we define new performance metrics for transactional request and present our algorithm. Section 4 we discuss our simulation results.

2 On-demand System Architecture

In this section we explain the system model. Figure 1 shows the architecture of a typical on-demand broadcast system [DirectPC]. There is a single server that supports a large client population. When a client needs a data item that it cannot find locally, it sends a request for the item to the server through uplink channel. These request are queued up at the server on arrival. The server chooses an item based on the current status of the request queue, and broadcasts it over the satellite link, and removes the all the request for that data item from the request queue. The client monitors the broadcast to download the requested data item. Similar to previous work on broadcast scheduling [AF98, FZ96] we make following assumptions about the environment: We assume that there is a single broadcast channel that is monitored by all the clients. We do not consider transmission errors, so when a data item is broadcasted it is received by all the clients waiting for it. All the data items are assumed to be locally available on the server. For simplicity we

assume that all data items have equal size and hence have equal service time. Each data item is broadcasted entirely at once. The time it takes to broadcast a data item is referred to as broadcast tick (time to compose a broadcast plus the communication time) [AF98].

3 A Review of earlier Scheduling Algorithms and Motivation

A number of scheduling algorithms have been reported in the literature, however, majority of them are push-based. There are some algorithms [AF98, Xu97, WJ88, Dy86] that are for pull-based broadcast system and we review them here.

In [FZ96, ST97, VH96] scheduling algorithms for push-based broadcast setting are proposed. In these approaches the server delivers data using a periodic broadcast program, which is based on the estimation of access probability of each data item. Its usefulness is limited to static environment where access probabilities do not change often. In [Ac95] broadcast scheduling algorithm for hybrid push-pull environment are proposed where the server periodically broadcasts using a broadcast program. A part of channel bandwidth is reserved for data items, which are to be pulled by the client. The client issues a request to server only after it has waited for a predefined duration for data to become available in periodic push based broadcast.

Pull based scheduling algorithms FCFS (First Come first Serve), MRF (Most Request First), MRFL (Most Request First Lowest) and LWF (Longest Wait First) were studied in [WJ88, Dy86]. In FCFS, pages are broadcasted in the order they are requested. In MRF, page with the maximum pending request is broadcasted. MRFL is similar to MRF, but breaks ties in the favor of page with lowest request access probability. In LWF waiting time that is the sum of waiting time of all pending requests for the page is calculated for all the pages. The page with the longest waiting time is broadcasted. It is shown in [WJ88, Dy86] that FCFS performs poorly compared to other algorithms in broadcast environment when the access pattern is non-uniform. In [Xu97] authors studied on-demand systems where requests were associated with deadlines. They reported that on-demand broadcast with EDF (Earliest deadline First) policy performs better. In [AF98] authors proposed a scheduling algorithm, referred to as $R \times W$ (R stands for number of pending requests and W stands for waiting time of the arrival of the first request) that combines FCFS and MRF heuristics. The algorithm computes the product of R and W and selects the data item with the maximum $R \times W$ value for the next broadcasting. In these previous on-demand scheduling algorithms the scheduling decisions were taken at every broadcast tick. It has two implications:

- First, decision overhead that is cost of making a scheduling decision became a very important factor. In order to make full use of broadcast bandwidth the time required to make broadcast decision must be less than the length of a broadcast tick. A scheduler that makes decision slowly will stall the broadcast, resulting in unused bandwidth, thereby wasting a critical resource. This could happen in a large-scale dynamic system, with large database size and high request rate. In such a system scheduler of the previous algorithms would have to scan a large number of request queue entries before making a decision with in a broadcast tick. Also broadcast tick

could be small, in case of a smaller size of broadcast data item or if the channel is of high bandwidth. We show this using a mathematical formulation below.

Let N be the size of the database. Let S_i be the size of the data item D_i. Let B be the broadcast bandwidth. Let T be the time taken for scheduling decision for the deciding the data item to be broadcasted. T is commonly refereed to as scheduling overhead.

The time take to broadcast the data item is given by $Tick = S_i / B$.

For efficient use of broadcast bandwidth $T < Tick$. In most scheduling algorithm [AF98, WJ88, Dy86, VH96] T is of the order of O(N). Broadcast Bandwidth will be wasted when $T > Tick$. This can happen in following scenarios:

- S_i is small. (Size of the data item to be broadcasted is small). This would make *Tick* smaller. Also not that size of the data item has no effect on the scheduling decision time.

- B is large. (Broadcast bandwidth is large)(Hughes DirectPC provides 400Kbps of downlink bandwidth, Wireless LANs provide up to 10Mbps bandwidth). This would also make *Tick* smaller.

- N is large. (Size of the database is large). This would make T scheduling overhead larger.

- Second, since the scheduling decision is made at every broadcast tick, consequently, scheduler cannot predict the data items to be broadcasted in immediate future broadcast ticks and there is no notion of periodicity and broadcast cycle. Hence clients have to continuously monitor the broadcast to download their requested data items. Mobile clients because of power constraints may not afford to continually monitor the broadcast. With the use of index the mobile client can check if its required data is present in the current broadcast cycle and if it is not then the mobile client can go to sleep and tune in the next broadcast cycle for index.

We propose an on-demand scheduling mechanism, which takes scheduling decisions at periodic interval. This gives a periodic nature to the broadcast and also indexing can be used and we avoid the problem of decision over head per broadcast tick. Also if client is locally caching the data item for faster access, the server can do periodic updates and inform about the updates to the client at the end of the broadcast cycle. The client can check at the end of the broadcast cycle if its locally stored data item has been updated at the server thus avoiding continuous monitoring of broadcast.

To the best of our knowledge this is the first time in literature that periodic scheduling has been explored for on-demand system. We show using simulation that taking decision at periodic interval does not degrade the access time of the data item and also show that

our algorithm performs better than the previously proposed FCFS, MRF and R*x*W algorithms.

4 Our Algorithm

We now describe our broadcast scheduling algorithm, which has low overhead, low complexity, scalable and provides excellent performance across a range of scenarios. In previous works [AF98, AM98, Ka01] LWF has been discarded as high overhead, impractical algorithm for large system. This is because it calculates total accumulated wait time for every data item with pending requests in order to decide which data item to broadcast. For large scale dynamic on-demand system there will be large number of pending requests for many data items and also there may be at least one pending request for every data item in the database. As a result a for a system of high bandwidth with large database, the LWF scheduler may not take scheduling decisions within a broadcast tick resulting in unused bandwidth. LWF has been shown to be a bottleneck in [AF98] with broadcast bandwidth of 155 Mbps with database size of 5543 8K pages. We propose a new measure that we define as Approximate Wait Time. It is based on LWF, however has low computation overhead and has performance similar to LWF and performs better than all the previously proposed algorithms (FCFS, MRF and R*x*W) for all the cases.

4.1 Approximate Wait Time (AWT)

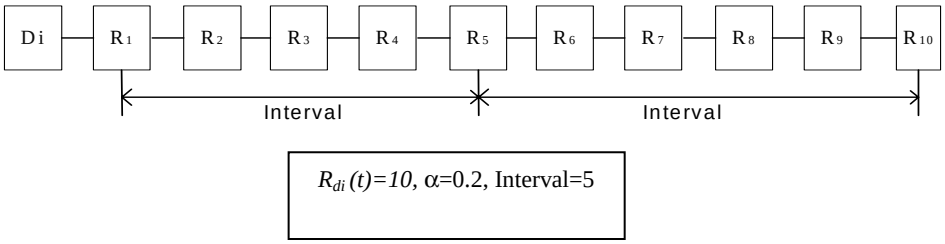


Fig. 2 Approximate Wait Time

Approximate Wait Time is an estimate of Total waiting time of pending requests for a data item. For calculating approximate wait time we use parameter *depth (d)*. *Depth* is defined as the number of request entries to be scanned for the data item for calculating AWT. *Depth* is directly proportional to the number of request for the data item d_i, $R_{di}(t)$ at time instant t. Depth is calculated using following equation

$$d = \alpha \times R_{di}(t) \text{ ; where } (0<\alpha<1) \qquad (1)$$

We use the parameter α as the measure of overhead and accuracy for calculating AWT. Higher the value of α the more closer will be AWT to the actual total wait time but the scheduling overhead will increase considerably as number of request entries to be

scanned increases. When α=1, the approximate wait time will be the actual total wait time of all the outstanding request of the data items. *Interval (T)* is defined as $R_{di}(t)/d$. Interval, T, denotes that every T^{th} waiting request for the data item is scanned. For the example shown in figure 2 $R_{di}(t)=10$, α=0.2, depth (d) is calculated using equation (2) as (0.2×10) 2. Depth denotes that two entries will be scanned for calculating AWT. Now Interval is 5(10/2). Hence R1 and R5 and R10, the last entry, are scanned. AWT is evaluated using the following equation.

$$AWT(t) = R_{di}(t) \times \left(\left(\sum_{k=0}^{d-1} t - A_{i \times T} \right) + (t - A_{Rdi(t)}) \right) / d \qquad (2)$$

4.2 Periodic Approximate wait time algorithm

We have developed a new scheduling algorithm called PAW (Periodic Approximate Wait) which gives better performance than FCFS, MRF and R$\times$W algorithm and has very low overhead compared to LWF and all the other algorithm.

PAW maintains a request queue with a single entry per data item that has outstanding request. Each entry in the queue contains data item identifier (d_i), total number of outstanding request (R_{di}), timestamps of all the requests for that data item. The timestamp are maintained in their order of arrival. AWT value for an entry is calculated using equation 2. When a request arrives at the server, the server performs a look up to find the entry in the Request queue for the requested data item. If the entry for that data item is found in the request queue then R_{di} is incremented and time stamp of its arrival is saved. If that request were the first request for that data item there would be no entry in the request queue. Hence a new entry is created for that data item in the Request queue, R_{di} is initialized to 1 and the timestamp of arrival of the request is stored in that entry. Note that the request queue size is bounded by N, number of data items in the database.

In our algorithm we take scheduling decisions at periodic intervals, which we refer to as *broadcast cycle*. We use the notion of broadcast cycle for introducing periodicity in broadcast. Broadcast cycle concept has been used in scheduling algorithms of push-based system [FZ96, Ac95, Li02, ST97] but has not been used in on-demand system. In push-based system the content and organization of a broadcast cycle (referred to as schedule) is same in every cycle and the same schedule is repeatedly broadcasted. However, unlike push-based system, in our scheme content and organization of each broadcast cycle may vary depending on the current workload at the server. We use broadcast cycle notion for dual purpose: Firstly, for introducing periodicity in broadcast so that indexing can be used. Secondly, it is used as interval after which updates at the server can be communicated to the client. So client can tune in the broadcast channel at

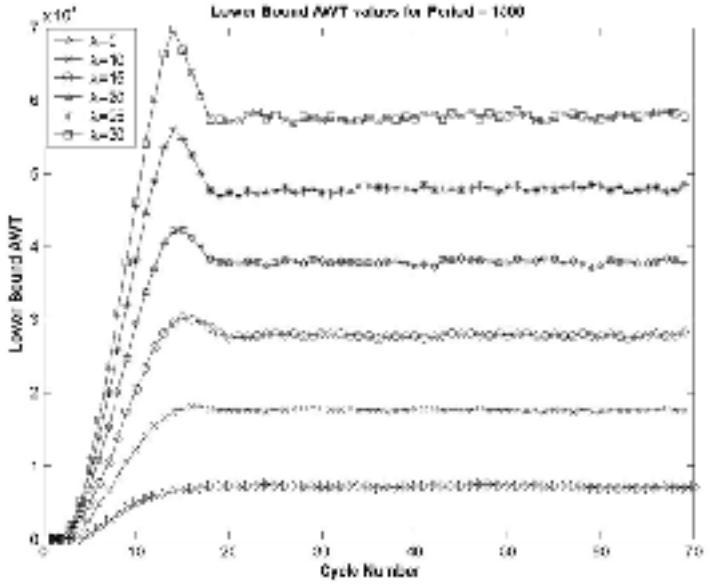


Fig. 3 AWT value of the last data item of the period

regular intervals and check if any of their cached items are invalidated due to the updates at the server. Broadcast cycle is an interval of length *K* broadcast ticks.

At periodic intervals of K broadcast ticks, where K is the period of broadcast, the server selects data items to be broadcasted in the next K broadcast ticks. The server calculates AWT values for all the request queue entries. The request queue entries are sorted in descending order based on their AWT values. The server selects first K entries from the sorted request queue to be broadcasted in the next K broadcast tick. The K data items are broadcasted in the order they were sorted. The entry for the data item is removed from the request queue once the data item is broadcasted. The overhead for our algorithm is *O(NlogN)*. Note that the previous algorithms have *O(N)* overhead at every broadcast tick. So in the case of large database, N would be large, when the system has high bandwidth or size of the data item is small the server would take more time to take the broadcast decision than the duration of broadcast tick. In our case we take scheduling decision at every K broadcast tick hence there is no risk of wasted broadcast bandwidth at every tick as is the case in all the previous on-demand broadcast algorithms.

While performing experiments we observed that the average AWT value of the K^{th} data item in the sorted request queue typically converges to some value. Figure 2 shows the AWT values of the last data item in the period for all the cycles for different request rate.

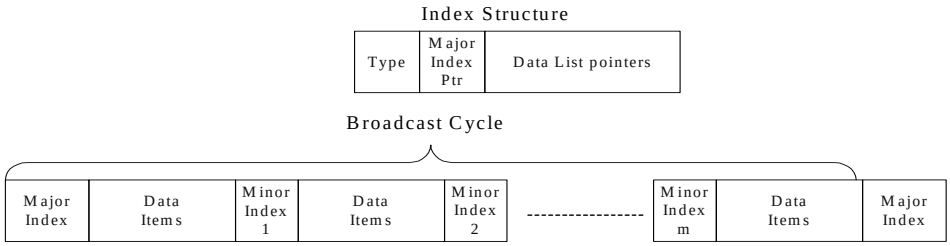


Fig. 4 Indexing Structure

We observe that after the initial warm up period the AWT values of the last data item in the period stabilizes to a constant value. Figure 2 shows the AWT value of the 2500th and 500th item in the sorted request queue for period of 2500 and 500 respectively. In [AF98] they had observed that highest RxW value converges to a some value after steady state. However in our experiments we observed that every element of the sorted request queue converges to some constant value when AWT is used as a heuristic. This insight can be used to avoid searching the entire list of pages with outstanding requests. We maintain the running average of the AWT value of the Kth data item (that is the AWT value last data item in the period). We refer to this running average as *Lbound* (read as Lower bound). We use value $\theta \times Lbound$ for comparison with the AWT value of the queue entry. We refer to θ as the pruning factor and ($0< \theta <1$). After every broadcast period, when calculating AWT value for an request queue entry if the AWT value is greater the $\theta \times Lbound$ then entry is copied in a separate Sort list. Higher the value of θ more closer will be the size of the sort list to K (size of the broadcast period). A Lower value of θ would increase size of the sort list greater K and when $\theta = 0$ the size of the sort list will be equal to that of the request queue. The sort list is sorted with respect to AWT value in descending order. And the first K data items are selected for broadcast in the next broadcast cycle. This reduces the overhead the overhead of our algorithm to *O (KlogK)*. The *Lbound* is initialized to 1 and calculated for cycle n+1 using the following equation

$$Lbound(n+1)=(Lbound(n)+Awt(P))/2$$

Where *Lbound(n)* is the *Lbound* value of the nth cycle and Awt(P) is the Awt value of the last element to be broadcasted in a period whose value is P broadcast ticks.

4.3 Indexing

Index is a directory of the list of data items, which are to be broadcasted in the broadcast cycle. We adapt (1, m) indexing [Im94] mechanism for our algorithm. In this method entire index is broadcasted at the beginning of every broadcast cycle and then after every (*K/m*) broadcast slots, where *K* is the length of the current broadcast cycle. We refer to the index broadcasted at the beginning of every broadcast cycle as *major index* and index broadcasted inside broadcast cycle after every (*K/m*) broadcast slots as *minor index*. The indexing structure is shown in figure 4. All indexes identify themselves, whether it is a

Table 2. Simulation Parameters and settings.

Symbol	Description	Default
λ	Mean request arrival rate (Exponential)	10 requests/tick [5-30]
DBSize	Database Size	10000
Period	No. of data items in a broadcast cycle	500 [100-2000]
offset	Shift in hot pages	0[0-2000]
α	Accuracy factor	0.1
θ	Pruning factor	0.9

major index or a minor index and contain pointer to the location of the next major index. The *minor index* contains the list of data items that are not yet broadcasted in the current broadcast cycle. In push based system there is no concept of minor index. All indexes in the push-based system are

of the same size and contain list of next *K* element to be broadcasted. In our algorithm the i^{th} minor index within the cycle, will contain the list of $(K-i\times(K/m))$ data items yet to be broadcasted in the cycle. The major index and all minor indexes that are broadcasted are stored at the server until the end of the current broadcast cycle.

4.4 Client Side

Client sends request for data item to the server if it is not available locally. After sending the request client tunes to the broadcast channel for downloading the next index to be broadcasted. If the client gets the data item before the next index then it downloads the data item and does not wait for the index. The client checks if required data is present in the current index and if it is then it tunes in the current cycle to download its required data item. If the data is not present in the index then the client sleeps for the current broadcast cycle and tunes in for the next major index that would be broadcasted for the next cycle.

5 Experimental Results

5.1 Performance metrics

We have used following performance metrics for our proposed algorithm:

Response Time of a request: Response time is a most common measure of any scheduling algorithm. It is the difference in time when the request is sent to the server and the time when the data item is broadcasted.

Tuning Time: It is the total time spent by the client listening to the broadcast channel. Listening to the broadcast channel requires the client to be in active mode. Hence, the tuning time for accessing data determines the amount of time spent by the mobile unit in active mode. Tuning time is the measure of the power consumed by the client to retrieve

the required data. In case of the previous on-demand algorithms, there is no provision for creating index and their tuning time is equal to the total wait time of the transaction. Our scheduling algorithm has provision for broadcasting index and reduces client-tuning time.

Worst case waiting time: It is defined as the maximum amount of time that any user request waits before being satisfied. This performance metric if an algorithm is causing starvation for any request. This is an important criterion for interactive applications.

Robustness: Robustness of an algorithm indicates that the algorithm performs well in presence of dynamically changing environment. We test this by changing in the request pattern of the user requests. We perform two experiments. First we change the request arrival continuously and test the sensitivity of the algorithm to change. Second we change the data access pattern, in this experiment we change the range of hot pages during the experiments.

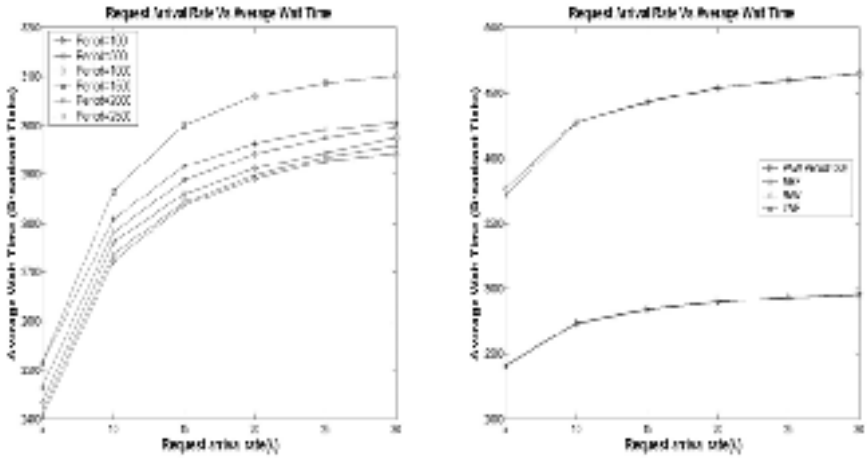
5.2 Simulation Environment

We used simulation model written in CSIM [Sc86] to compare its performance with other algorithms. The model represents environment similar to that described in Section 3. The broadcast channel is modeled as a server with fixed broadcast rate. We do not specify an absolute value for this rate but use *broadcast tick* to measure simulated times. This approach emphasizes that the results are not limited to any particular bandwidth and/or data item size but describe tradeoffs among algorithms. Scheduling overhead is not included in the results here. In other words we assume that all the algorithms are able to make scheduling decisions fast enough to keep the broadcast bandwidth fully utilized. In the model, the client population is represented by a request stream. We use an open system model since our work is aimed at supporting large dynamic client populations, such client populations cannot be modeled with a closed simulation system. The cost of using the back channel for sending the transaction request to the server is small and hence not modeled.

The main parameters and settings for the workloads used in the experiment are shown in Table 2. The client population model generates requests with exponential inter-arrival times with mean λ. The access pattern is shaped with 80:20 [Gr94] distribution rule. That is 80 percent of the request access 20 percent of the data items. The requests are distributed over a database containing *DBSize* fixed-size pages.

5.3 Experiments

5.3.1 Average Waiting Time



(a) **(b)**

Fig. 5 Average waiting Time

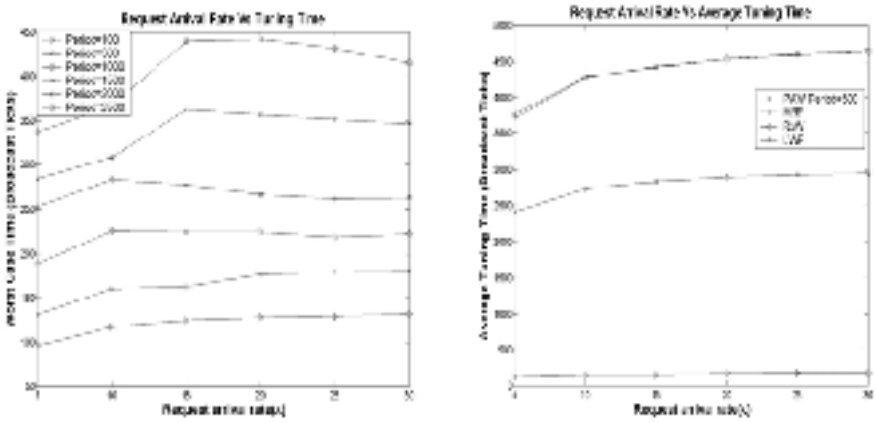
In this experiment average waiting time is measured as the request arrival rate is varied from 5 to 30 requests per tick. Figure 5b shows the average waiting time for each MRF, R×W, PAW with period = 500, AWT. For each algorithm it observed that average waiting time increases initially but levels of after the request arrival rate of 15. As depicted in 5b with a period of 500 broadcasts tick our algorithm results in 36% reduction in Average wait time compared to R×W and MRF algorithms. Also we observe average wait time of our periodic algorithm is similar to the AWT algorithm when the scheduling decision are taken at every broadcast tick.

Figure 5a shows the average waiting time of our PAW algorithm for periods [100, 500, 1000, 1500, 2000, 2500]. It was observed that average wait time increases with increase in the broadcast period. We observed that at period = 2500, the average waiting time is not more than 5% worse than that with period 100. Also up to period=1000 the difference among the average waiting times is no more than 0.6% worse than the PAW with lower period. Also waiting time with period=2000 and 2500 is not more than 1.8% and 5% worse compared to AWT. Thus we reduce the scheduling overhead and risk of wasting broadcast bandwidth without any considerable penalty of average waiting time.

5.3.2 Average Tuning Time

In this experiment average tuning time is measured as the request arrival rate is varied from 5 to 30 requests per tick. Figure 6b shows the average tuning time for MRF, R×W,

PAW with period = 500, AWT. Compared to MRF and R×W our PAW algorithm with period=500 achieves 96% reduction in client tuning time and compared to AWT algorithm we achieve 93% reduction in client tuning time. Figure 6a shows that average client tuning time per request increases with increase in the broadcast period. The tuning



(a) **(b)**

Figure 6. Average Tuning Time

time increases from 100 ticks to 440 ticks as broadcast period is increased from 100 to 2500. As the size of the broadcast cycle (broadcast period) increases the interval among the index increases. So the client has to tune for more number of ticks at to monitor index. As a result there is increase in client tuning time.

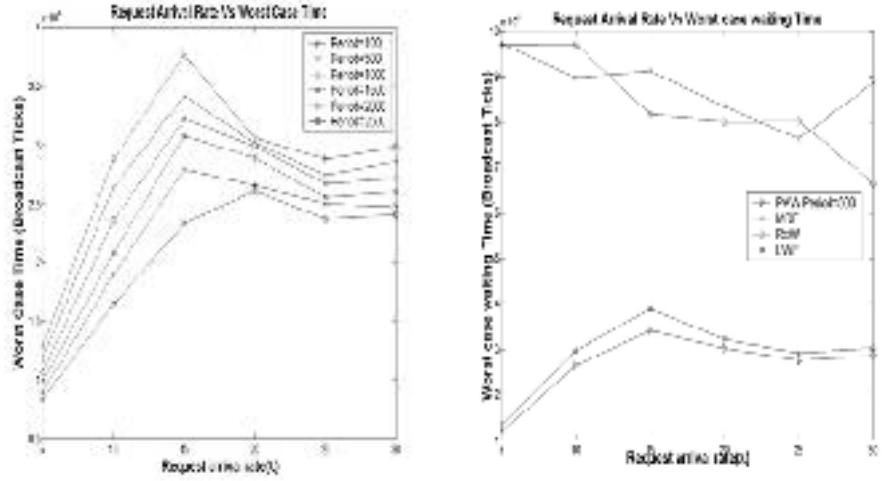
5.3.3 Worst case waiting time

In this experiment worst case waiting time is measured as the request arrival rate is varied from 5 to 30 requests per tick. Figure 7b shows the Worst case waiting time for MRF, R×W, PAW with period = 500, AWT. The longest measured waiting time for any request during the simulation run is plotted. At higher request rate =30, PAW with period 500, has worst case waiting time 56% less than RW R×W and has 68% less than MRF. At lower request rate =10, PAW with period 500 has worst-case time 70 % less than that of R×W. Also our algorithm has worst-case time about 12% less than when scheduling decision is taken at every broadcast tick.

Figure 7a shows the average waiting time of our PAW algorithm for periods [100, 500, 1000, 1500, 2000, 2500]. It was observed that worst-case wait time deceases with increase in the broadcast period. There is a peak visible at in worst case waiting time when request arrival rate = 15 after which the worst-case wait time decreases and levels off. The difference among the worst-case time for the periods is not more than 15%.

5.3.4 Robustness

In this experiment we examine the impact of interest shift in varying frequency on the average wait time. To model shift in interest in this experiment we shift the hot spot by



(a) **(b)**

Fig. 7 Worst case waiting time

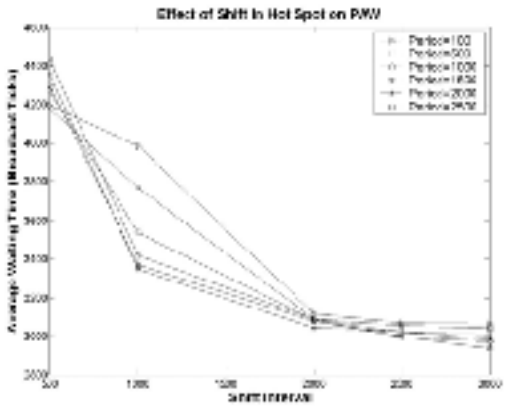


Fig. 8 Effect of shift in hot spot

1000 pages after every *shift interval*. The *shift interval* is varied from 500 to 3000. Figure 8 shows the Average waiting time for different periods of our PAW algorithm when the shift interval is varied.

6 Conclusion

In this paper we have studied the problem of scheduling multiple items and transactional requests in an on-demand broadcast environment and proposed a scheduling algorithm for managing such requests. Previous works in this context on single item request assumed prior knowledge of static client access pattern, which may not always be the case in reality. Hence our work is not only complimentary to the previous works in this area but discovers new insight in information dissemination. We showed through simulation that our algorithm performs better than the common single item scheduling algorithms. At present we are working on the optimization problem for dynamically determining the optimal size of broadcast cycle period.

References

[Ac95] Acharya Swarup, Rafael Alonso, Michael Franklin, and Stanley Zdonik. "Broadcast Disks: Data Management for Asymmetric Communication Environments". In Proceedings of ACM SIGMOD Conference, CA, 1995.

[AF98] Aksoy, D., and Franklin, M. Scheduling for Large-Scale On-Demand Data Broadcasting. In Proceedings of IEEE Infocom, CA, 1998.

[AM98] Acharya Swarup and Muthukrishnan S. Scheduling on-demand data broadcasts: New metrics and algorithms. In Proc. of Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1998.

[DirectPC] Hughes Network Systems. DirecPC Home Page. http://www.direcpc.com, Jan, 2001

[Dy86] Dykeman, H. D., M. H. Ammar, and J. W. Wong, "Scheduling Algorithms for Videotex Systems under Broadcast Delivery", Proc ICC'86, 1986, pp. 1847-1851.

[FZ96] Franklin M, Zdonik S, "Dissemination-Based Information Systems ", IEEE Data Engineering Bulletin, 19(3), September, 1996.

[Gr94] Gray J, Sundaresan. P, Englert. S, Baclawski. K,Weinberger P. "Quickly Generating Billion-Record Synthetic Databases" Proc. ACM SIGMOD Conf., Minneapolis, MN, May, 1994.

[Hr87] Herman G, Gopal G., Lee. K., and Weinrib. A. “The Datacycle architecture for very high throughput database systems”. In Proceedings of ACM SIGMOD, CA, 1987.

[Im94] Imielinski, S. Viswanathan, and Badrinath B. R.. "Energy Efficient Indexing On Air". In Proceedings of ACM SIGMOD Conference, 1994.

[Ka01] Murat Karakaya , "Evaluation of a Broadcast Scheduling Algorithm", Lecture Notes in Computer Science, 2151, 2001.

[Li02] Liberatore Vincenzo, "Multicast Scheduling for List Requests", Proceedings of IEEE Infocom, CA, 2002.

[Sc86] Schwetman, H. “CSIM: A C-Based, Process-Oriented Simulation Language”, Proceedings of Winter Simulation Conference, 1986.

[SK97] Stathatos. K, Roussopoulos. N and Baras J. S., "Adaptive Data Broadcast in Hybrid Networks ", Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997.

[ST97] Su. S, Tassiulas. L, "Broadcast scheduling for information distribution", INFOCOM, 1997.

[VH96] Vaidya N and Hameed. S. “Data broadcast in asymmetric wireless environments”. In First International Workshop on Satellite-based Information Services (WOSBIS), 1996.

[WJ88] Wong. J “Broadcast Delivery”. In Proc. of the IEEE, 76(12), 1988.

[Xu97] Xuan, Sen, Gonzalez, Fernandez, Ramamritham. “Broadcast on-demand: Efficiently and timely disseminating of data in mobile environment”. IEEE RTAS’97.