RC 20621 (11/11/96)
Subject Area Mathematics/Computer Science

IBM Research Report

The Maximum Factor Queue Length Batching Scheme for Video-on-Demand Systems

Charu C. Aggarwal, Joel L. Wolf and Philip S. Yu

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

The Maximum Factor Queue Length Batching Scheme for Video-on-Demand Systems

Charu C. Aggarwal, Joel L. Wolf and Philip S. Yu

IBM T. J. Watson Research Center

Yorktown Heights, NY

December 10, 1996

Abstract

In a video-on-demand environment, batching of video requests is often used to reduce I/O demand and improve throughput. Since viewers may defect if they experience long waits, a good video scheduling policy needs to consider not only the batch size but also the viewer defection probabilities and wait times. Two conventional scheduling policies for batching are the first-come-first-served (FCFS) policy which schedules the video with the longest waiting request, and the maximum queue length (MQL) policy which selects the video with the maximum number of waiting requests. Neither of these policies lead to entirely satisfactory results. MQL tends to be too aggressive in scheduling popular videos by considering only the queue length to maximize batch size, while FCFS has the opposite effect by completely ignoring the queue length and focusing on arrival time to reduce defections. In this paper, we introduce the notion of *factored queue length* and propose a batching policy that schedules the video with the *maximum factored queue length*. We refer to this as the *MFQL* policy. The factored queue length is obtained by weighting each video queue length with a factor which is biased against the more popular videos. An optimization problem is formulated to solve for the best weighting factors for the various videos. We also consider MFQL implementation issues. A simulation is developed to compare the proposed MFQL variants with FCFS and MQL. Our study shows that MFQL yields excellent empirical results in terms of standard performance measures such as average latency time, defection rates and fairness.

1 Introduction

Recent advances in communications technology have created a considerable amount of interest in video-on-demand (VOD) systems. In such a system, viewers have the flexibility of choosing both the video they want as well as the time at which they wish to watch it. Such a system can be implemented using a client-server architecture. A large set of videos is stored in a centralized server. The clients consist of viewers who make requests from the server. Whenever there is a request for a particular object, the video is accessed from the disks in the storage server and transmitted to the viewer isochronously.

Because of the isochronous requirement, each disk or disk array can only be allowed to support some fixed number of concurrent video streams [17, 16]. Hence a constraint exists on the maximum number of video streams which can be supported concurrently by a video server. This maximum is referred to as the server *stream capacity.* If the number of outstanding viewer requests is larger than the available server stream capacity, a new viewer request may not be satisfied immediately. The period which elapses between the arrival of a request and the time when the service to the display device is actually initiated is known as the *latency* of the request. An issue of concern is that a viewer may *defect* from the queue if the wait time becomes excessive. The goal is to provide good quality of service with little defection while consuming the least amount of server capacity. One way to accomplish this goal involves developing techniques which handle multiple requests using a single video stream.

A common approach to reducing video stream requirements is known as *batching.* The method of batching was proposed originally in [1]. We shall focus on batching in this paper. The basic idea is to delay the requests for the different videos for a certain amount of time so that more requests for the same video arriving during the current *batching interval* may be serviced using the same stream. Thus, requests which are made by many different viewers for the same video can share a common video stream if these requests are spaced closely enough. We assume that the viewer has a choice of selecting from a relatively large number of videos. Consequently, whenever a stream becomes available at the server end, a question arises as to which video is best to schedule at that particular moment in time. There has been a considerable amount of research devoted to scheduling policies for the batching technique [4, 6, 15]. One common approach to implementing batching is the first-come-first-served (FCFS) policy, which schedules the video with the longest waiting request. A second common approach is the maximum queue length (MQL) policy, which selects the video with the maximum number of waiting requests. Neither of these two lead to entirely satisfactory results. MQL tends to be too aggressive in scheduling popular videos by considering only the queue length to maximize batch size, while FCFS has the opposite effect by completely ignoring the queue length and focusing on arrival time to reduce defections. It was shown in [4] that the FCFS policy may be desirable to the MQL policy in many cases. Several improved heuristics have also been studied in [15].

In this paper we introduce the notion of *factored queue length* and propose a batching policy that schedules the video with the *maximum factored queue length*. This is referred to as the *MFQL* policy. The factored queue length is obtained by applying discriminatory weighting factors to the queue lengths of the different videos, where the weight decreases as the popularity of the video increases. An optimization problem is formulated to solve for the best weighting factors for the different videos. Our analysis shows that the optimal weighting factor is inversely related to the square roots of the relative request frequencies of the different videos. We also consider several variants of MFQL, in order to ensure implementability. A simulation is developed to compare the proposed MFQL policy with FCFS and MQL. Our study shows that MFQL yields excellent empirical results in terms of standard performance measures such as average latency time, viewer defection probability and fairness.

We briefly comment on some alternative techniques for reducing stream requirements. These include *bridging* and *adaptive piggybacking*.

The idea behind bridging involves the use of a buffer space in memory to retain moving portions of certain videos behind them as they play. Suppose that a contiguous length (in time units) of a particular video is being buffered as it plays for a particular viewer. Then a viewer who is trailing the original viewer by an amount of time less than this length can be serviced from this buffered stream instead of from disk. More details on the technique of bridging may be found in [10] and [18]. The technique of bridging has the potential disadvantage that a considerable amount of buffer space may be required in order to achieve any significant savings in server capacity.

Recently, a technique called adaptive piggybacking has been proposed [7, 8]. In this approach the display rates are altered even while the requests are in progress. If two streams for a common video are a small enough interval apart, the first stream can be played at a slower rate than the second. Ultimately, the second stream will catch up and can be piggybacked on the first. It has been established that small differences in the display rates (for example, those which deviate at most 5% from the normal display rate) are not perceived by the viewer. Nevertheless, piggybacking has potentially troublesome implementation issues when MPEG-like compression is used, because of inter-frame dependencies.

Finally we note that stream sharing increases the complexity of supporting VCR functions such as pause-resume. When batched or piggybacked viewers pause, additional stream capacities may need to be allocated to support the subsequent resume. Different approaches to support VCR functions can be found in [18, 5, 3]. In [18], a novel scheme using look-ahead streams with buffering is devised to guarantee immediate resume under batching. Efficient support of VCR functions is an important subject on its own right, but is orthogonal to the focus of the current paper.

This remainder of this paper is organized as follows. Definitions of the important performance metrics are given in the next section. The optimization model, its solution, and the implications regarding optimum batching policies are presented in Section 3. In Section 4, we discuss issues regarding the implementation of MFQL. Empirical results from simulations are described in Section 5. Section 6 contains a conclusion.

2 Performance Metrics

Consider a VOD system in which the server contains a database of N different videos from which the clients may make requests.

In our analysis of scheduling policies, we use the following performance measures:

- *Defection probability*: This is the probability that an arriving viewer leaves the system without being served, due to a wait time which exceeds the viewer's tolerance. We note that minimizing the defection probability maximizes the system throughput.

- *Average latency time*: The latency of a viewer is the period which elapses between the arrival of the video request and the time when the service to the display device is actually initiated. For obvious reasons, only non-defecting viewers can be considered in obtaining the average latency time measure.

- *Unfairness*: Let d_i denote the defection probability for video i, so that $\overline{d} = \sum_{i=1}^{N} d_i/N$ is the mean defection probability. Then the unfairness is equal to $\sqrt{\sum_{i=1}^{N}(d_i - \overline{d})^2/(N-1)}$. The fairness is inversely related to this standard deviation: Higher values correspond to less fair policies.

3 The Optimization Model

In this section we present a simple optimization model in order to minimize the average latency time subject to the constraint on server stream capacity. We stress here that the purpose of building this idealized model is to provide intuition about the key aspects of the problem, so that we can glean heuristics which will work well in practice.

Assume that the frequency of the ith video is known, and denoted by f_i. If L_i denotes the length of video i, then $L = (\sum_{i=1}^{N} f_i \cdot L_i)/(\sum_{i=1}^{N} f_i)$ is the average video length. Assume that the server capacity (in terms of the number of streams) is S. Consequently, the average number of streams which are scheduled by the server per unit time at full capacity is S/L. Suppose that $t_1, t_2, \ldots,$ and t_N are the *average time intervals* at which the videos $1, 2, \ldots,$ and N are batched. Then the average latency for a video of type i is equal to $t_i/2$. Our objective in this model is to minimize the average latency of the viewers. Thus, modulo a constant which we can ignore, we want to minimize the sum of the latency times of all the requests which arrive in a unit interval of time. The expected number of such requests for video i is f_i. Consequently, the expected sum of the latency times of all the requests which arrive within a unit interval is equal to $\sum_{i=1}^{N} f_i \cdot t_i/2$. This is the objective function we wish to minimize.

We need a constraint which limits the number of video streams which can be scheduled at any moment of time. On average, the number of streams for video i scheduled per unit of time is approximately equal to $1/t_i$. (This is not entirely rigorous, but if X is a random variable with a relatively small deviation, the approximation $E[1/X] \approx 1/E[X]$ is fairly accurate.) Thus, at full capacity the total number of streams of all video types scheduled per unit of time is $\sum_{i=1}^{N} 1/t_i$, which must be equal to S/L. Thus, we have the following constrained non-linear program:

$$\text{Minimize } \sum_{i=1}^{N} f_i \cdot t_i$$

$$\text{such that } \sum_{i=1}^{N} 1/t_i = S/L$$

$$\text{and } t_i > 0 \qquad \forall i.$$

The last constraint ensures the positivity of the variables. This is a simple constrained non-linear program. To solve it, we make a change of variables. Let $\rho_i = 1/t_i$. Then the objective function $\sum_{i=1}^{N} f_i/\rho_i$ becomes convex. The first constraint $\sum_{i=1}^{N} \rho_i = S/L$ becomes linear, and the positivity constraint becomes $\rho_i > 0$. This transformation thus yields a *continuous convex separable resource allocation problem*, and the optimal solution occurs when the derivatives of each f_i/ρ_i are equal [9]. For the benefit of readers unfamiliar with this fact, we derive it briefly: Because of the nature of the problem, the Kuhn-Tucker conditions [2] are necessary and sufficient for the existence of a global optimum solution. These Kuhn-Tucker conditions are as follows:

$$f_i - u/t_i^2 = 0$$
$$\sum_{i=1}^{N} 1/t_i = S/L.$$

Here u is the Lagrange multiplier associated with the constraint. From the above set of variables it is easy to see that the average batching intervals for the different videos should be related as follows:

$$t_1 \cdot \sqrt{f_1} = t_2 \cdot \sqrt{f_2} = \ldots = t_n \cdot \sqrt{f_n}.$$

(We have ignored the positivity constraints, but it can be seen that the optimum solution automatically satisfies them.) Note that the above equations hold when the derivatives of the convex functions in the transformed problem are equal, as observed previously.

In the absence of any defections from the system, the average queue length q_i of the ith video at the time of batching is equal to $q_i = t_i \cdot f_i$. Thus, using the above equations we get the following goals for the queue lengths at the time of batching:

$$\frac{q_1}{\sqrt{f_1}} = \frac{q_2}{\sqrt{f_2}} = \ldots = \frac{q_n}{\sqrt{f_n}}.$$

In other words, we have the following intuitive result:

Theorem 3.1 *In order to achieve the best average latency time, the queue lengths at the time of batching should be approximately proportional to the square roots of the relative frequencies of the arrivals of the different videos.*

Let the *factored queue length* of a video be defined as its queue length divided by the square root of its relative request frequency. The result of Theorem 3.1 automatically suggests a good *greedy* scheduling policy for the VOD problem as follows.

Whenever a stream becomes available, schedule the video with the largest factored queue length.

We call this the *Maximum Factored Queue Length* policy, or *MFQL* policy. Thus, the MFQL policy is closely related to the MQL policy, except that it weights the queue lengths with a factor biased against the more popular videos. We now comment on the differences in fairness of the MFQL and the MQL scheduling policies. Intuitively, the fairness of the system measures the variation in the quality of service provided across different videos. MQL is rather unfair to the cold videos, because the queue for a hot video tends to build up much faster than the queue for a cold video. The factoring of the queue lengths actually evens out the unfairness by working in favor of the cold videos. Thus, the MFQL policy not only optimizes the average latency time, but also provides much greater fairness when compared with the MQL policy. This result is serendipitous, since the MFQL policy was chosen quite independently of any fairness considerations.

4 Implementation Issues

MFQL attempts to ensure that the queue length at the time of batching is approximately proportional to the square root of the relative frequency of each video. To implement this policy, we need to address two important issues. The first is how to estimate the relative request frequencies to the different videos. The second is the implication of defections to the policy, which was derived under the assumption that there are no defections.

4.1 Estimation of Relative Frequency

A straightforward way of implementing MFQL is to divide the current queue length by the square root of a *moving window* estimate of the relative frequency. This estimate is obtained by counting the number of requests for each video occurring in a moving window of the immediate past period, for example a window of the past hour. Smoothing techniques can also be applied to take weighted averages over past estimates.

The problems with employing a moving window estimate are as follows. Even if the pattern of the demands for the videos changes only slowly over a period of time, the skewed request patterns make it hard to obtain statistically meaningful estimates for the request frequencies of the cold videos. Furthermore, in the case where the load varies quickly over time, even the relative request frequencies of the hot videos will be difficult to estimate accurately.

An alternative and apparently preferable technique for implementing MFQL involves measuring the *instantaneous* relative frequencies. The advantage of this method is that it works well, regardless of quick variations in the pattern of the demand over a period of time. Further, as we shall see, this technique leads to an interesting policy, which depends on the interval since a video was last scheduled.

For each of the videos $\{1, 2, \ldots N\}$, let Δt_i be the interval since the last time that video i was scheduled. Recall that q_i represent the length of the queue corresponding to video i. Because the value of f_i is not known robustly, we will approximate f_i by $q_i/\Delta t_i$. Using this substitution, the factored queue length is given by $\sqrt{q_i \cdot \Delta t_i}$. Since we need to choose the largest among this set of numbers, we may consider instead the square of these factored queue lengths for each video. Thus, another way of defining the factored queue length of video i is by multiplying its queue length q_i by Δt_i. In this way we obtain an implementable variant of MFQL, which we will call *instantaneous MFQL*, as follows:

Whenever a stream becomes available, schedule the video with the largest value of $q_i \cdot \Delta t_i$, where q_i is the queue length and Δt_i is the time since the last scheduling of video i.

4.2 Effects of Defection

To be realistic, a VOD system should be modeled as a loss queueing system: Customers may defect from the queue after waiting for a certain period of time. Thus the issue arises as to whether queue length or number of arrivals is the right metric to be used in the calculation of the factored queue length. The question here is whether we can simply ignore the customers who have already defected from the system while making the scheduling choice. If a larger percentage of the viewers corresponding to the cold videos defect anyway, then using the queue length instead of the number of arrivals may increase the unfairness of the system. Thus an alternative MFQL implementation approach is to substitute the number of requests for a video since the last time that it was scheduled for the queue length. (Of course, a selected stream should not be scheduled if *all* of the customers who arrived since the last time it was batched have defected.) Let a_i denote the number of such arrivals for video i, and consider the scenario in which the exact estimate of the relative frequency is used for the factored queue length calculation. The surrogate factored queue length becomes $a_i/\sqrt{f_i}$. Alternatively, we can define an implementable MFQL policy by combining this approach with the instantaneous concept. Thus we have the following scheme, called *instantaneous arrival MFQL*:

Whenever a stream becomes available, schedule the video with the largest value of $a_i \cdot \Delta t_i$, where a_i is the number of new arrivals and Δt_i is the time since the last scheduling of video i,subject to the constraint that the queue length q_i is positive.

Though this approach seems intuitively more plausible, we shall see in the next section that using q_i in the factored queue length formula actually leads to generally better emprical results than using a_i.

5 Simulation Model and Study

In this section we describe the empirical results which were obtained through simulations. In order to describe the experimental model, we first discuss details of the simulated viewer behavior. We assume that the video requests are generated by a Poisson process which has an interarrival time which is exponentially distributed with a mean of $1/\lambda$. An arrival corresponds to video i with probability $f_i/\sum_{i=1}^{N} f_i$, where f_i is the relative frequency of the video i. We shall assume that the relative frequencies are governed by the *Zipf* distribution [11]. Thus, we have $f_i = 1/i$.

Except for those few experiments in which customers were assumed to have infinite patience, we assume that the viewer may defect without being served if the latency time becomes too large. We model this user defection behavior by a normal distribution with a mean of μ and a standard deviation of σ. Note that we need to truncate the normal distribution on the left, because latency times can only be non-negative. However, if $\mu \geq 3 \cdot \sigma$, the error because of the truncation is negligible. In this paper, a mean of $\mu = 5$ minutes is assumed. This means that the user is willing to wait for a time x which is a random variable with a mean of $\mu = 5$ minutes and a standard deviation $\sigma = \mu/3$.

We assume that the system contains 100 videos, with an average length equal to 120 minutes. In order to simplify the simulations we do not consider support of any VCR-like functions such as pause or resume.

We consider first the idealized situation where the *exact* values of the frequencies are known, comparing the MFQL policy to other standard batching schemes. Clearly we

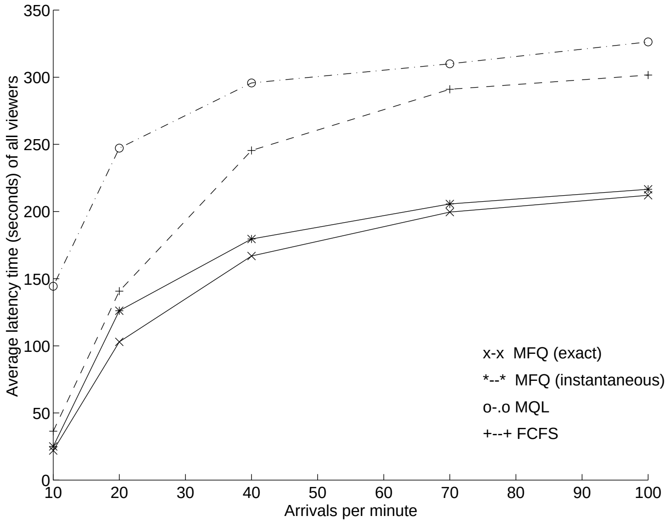

Figure 1: Average Latency Times in the Absence of Defections for Varying Arrival Rates

cannot "know" the exact frequencies in practice, but we simulated it anyway to use as a bound. Then we examined implementable MFQL variants. As we shall see, these idealized and realistic MFQL policies yielded relatively similar results.

5.1 Idealized MFQL Policy

The performance results of the various policies are depicted in Figures 1-8. In the figures, the Maximum Queue Length policy is denoted by MQL, and the First Come First Serve policy by FCFS. The MFQL policy assumes that the exact frequencies are known.

We first consider a scenario in which no defections occur. This will be instructive, in part because our optimization problem was modeled under this assumption. The average latency times of the MFQL, MQL and FCFS policies are illustrated in Figures 1 and 2. In the first figure, the arrival rate varies from 10 to 100 per minute, while the server capacity is fixed at 1000 streams. In the second figure, the server capacity varies from 500 to 2500 streams, while the arrival rate is fixed at 40 per minute. (The same pattern of parameters is repeated in subsequent experiments.) One striking aspect of this analysis is the relative performance of the FCFS and the MQL policies. It was shown in [4] that in terms of the average latency time, the MQL policy performs better than the FCFS policy. However, in the absence of defections, we see that the FCFS policy performs better than the MQL policy, even in terms of the average latency time. The explanation is that in measuring average latency time *only* the non-defecting viewers are considered. This biases the average latency time in favor of the MQL policy because the MQL policy has a much higher defection probability

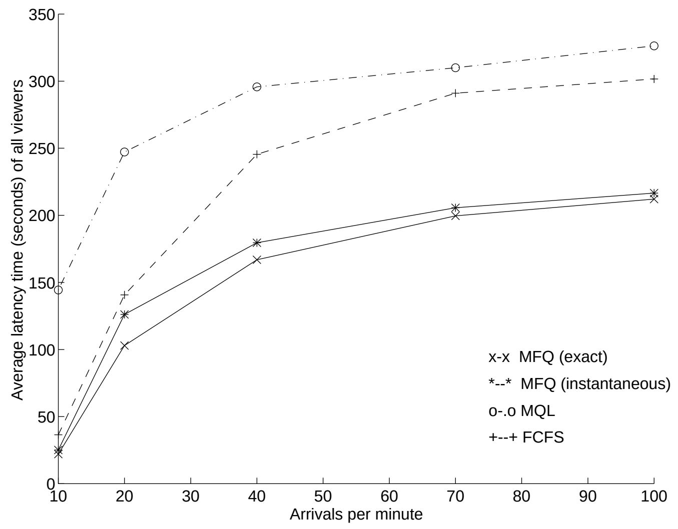

Figure 2: Average Latency Times in the Absence of Defections for Varying Server Capacities

and the defecting viewers would have a much higher average latency time had they stayed. (In fact, as we see shortly, the average latency time of the MQL policy is lower than even the MFQL policy when defections occur. Again, the reason is that the latency metric is misleading because the defections are factored out.) Finally, it is interesting to note in Figures 1 and 2 that MFQL has the lowest average latency of all.

In Figures 3-8 we consider more realistic cases incorporating viewer defections. We again performed simulations for cases where the arrival rate was fixed and the server capacity varied, as well as cases where the arrival rate varied and the server capacity was fixed.

The average latency times of the non-defecting users are illustrated in the Figures 3 and 4. The ordering is essentially clearcut throughout: MQL does better than MFQL, which does better than FCFS. Again, this metric can be somewhat misleading because it does not reflect the number of defecting viewers. MQL exhibits a scheduling bias, focusing on the popular videos at the expense of the less popular ones. This policy therefore encounters much higher defection probabilities. It is interesting to compare the dramatic difference between the pattern of average latency times in Figures 1 and 3, and similarly in Figures 2 and 4.

Figures 5 and 6 show the defection probabilities under these scenarios. As we see, the implementations of the MFQL policy do better than both the FCFS and MQL policies in terms of this metric. FCFS does reasonably well in cases of low arrivals or high server capacity. But MQL performs more or less consistently worse.

Another important performance metric is unfairness, which measures the difference in the quality of service provided to users requesting different videos. The relative

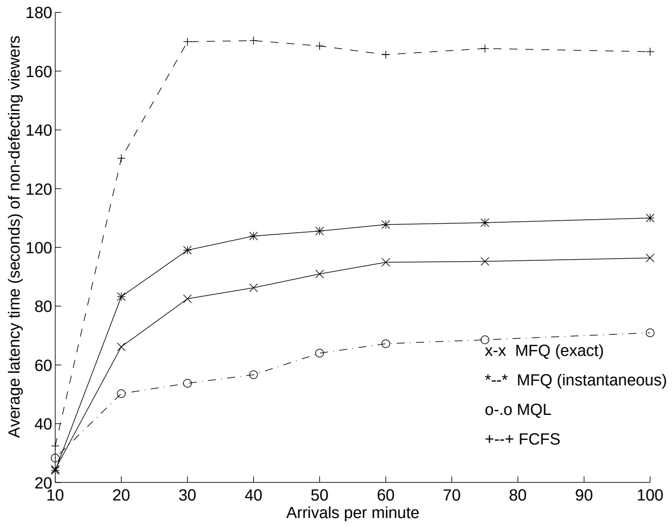

Figure 3: Average Latency Times for Varying Arrival Rates

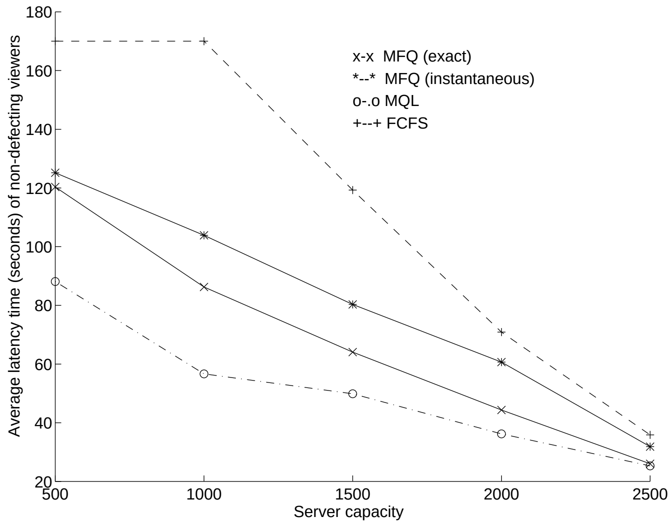

Figure 4: Average Latency Times for Varying Server Capacities

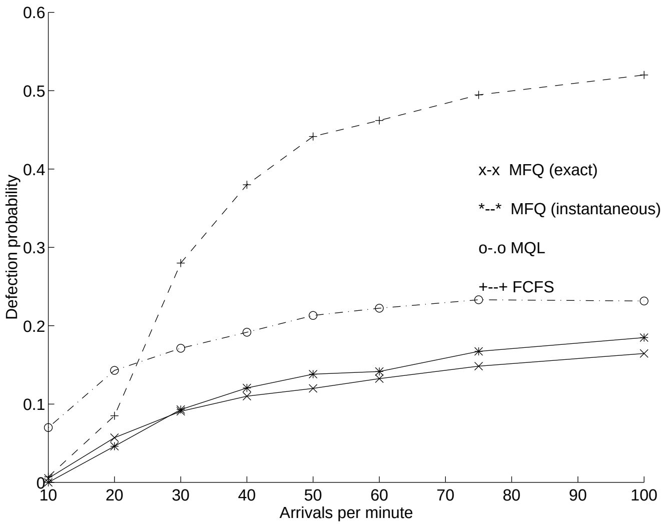

Figure 5: Defection Probabilities for Varying Arrival Rates

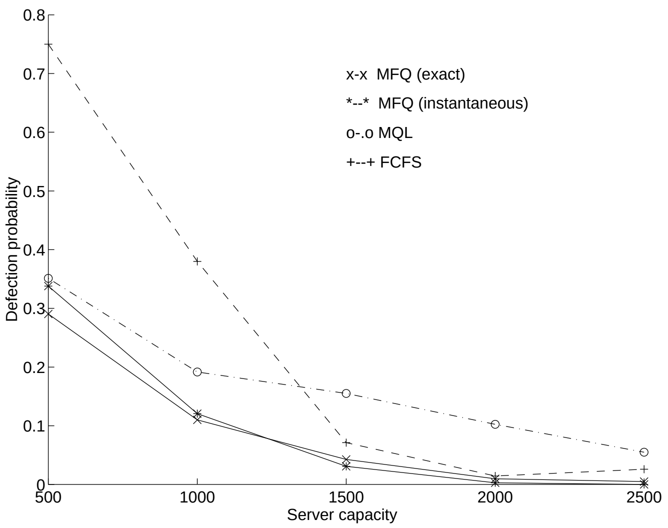

Figure 6: Defection Probabilities for Varying Server Capacities

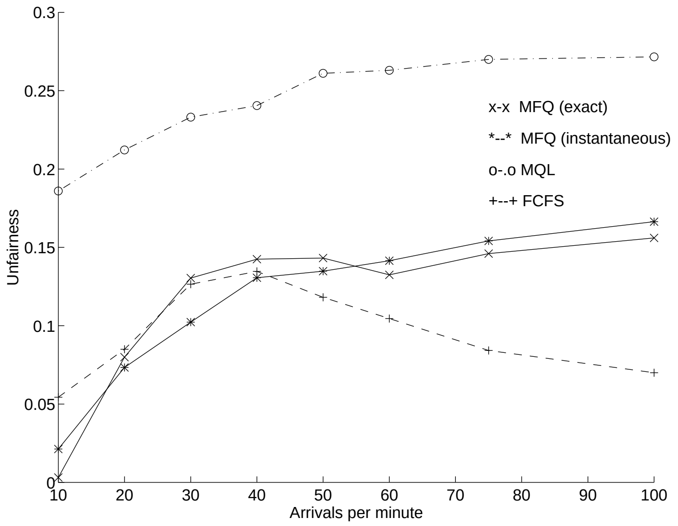

Figure 7: Unfairness for Varying Arrival Rates

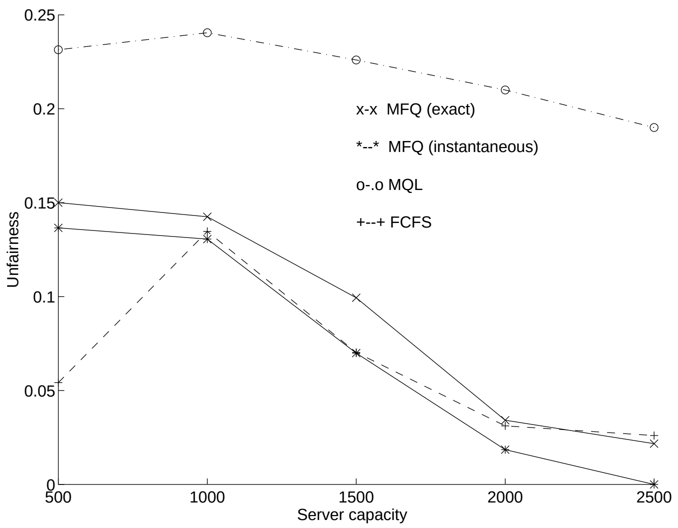

Figure 8: Unfairness for Varying Server Capacities

unfairness of the different policies are illustrated in Figures 7 and 8. In terms of unfairness, the MFQL policy performs surprisingly well: For low arrival rates and high server capacities, its performance is comparable to the FCFS policy. However, as the arrival rate increases, or the server capacity decreases, FCFS does become somewhat fairer than the MFQL policy. Nevertheless, the MFQL policy exhibits reasonable fairness, and is certainly much better than the MQL policy in this regard. In fact, one highly unpleasant property of the MQL policy we noted was that of *starvation*: Some of the less popular videos are never scheduled at all. This is because, for those videos, the viewers actually defect faster than they arrive! (To be more precise, when the queue length is strictly greater than zero, the defection rate is larger than the arrival rate. On an overall basis, however, the defection rate is virtually equal to the arrival rate.) The bottom line is that the queue length is essentially never large enough to allow the video to be scheduled via MQL.

5.2 Implementable MFQL Policies

Having seen that MFQL is a good scheme from a theoretical perspective, we now turn our attention to implementation issues. We have described two alternative methods for estimating relative frequencies, namely the moving window and instantaneous approaches. We have also suggested the possibility of substituting the number of arrivals for the queue length in the latter scheme.

We briefly compare the moving window to the instantaneous approach. As noted, we believe the performance of the latter is always as good as or superior to the former. (We shall omit descriptions of experiments which demonstrate these conclusions in the interest of space. But instantaneous MFQL outperformed moving window MFQL in each of the performance metrics.) Of course, moving window and instantaneous MFQL are merely two approaches to solving the same forecasting problem. A first advantage of instantaneous MFQL is that it does not involve any built-in parametric assumptions, and is therefore more straightforward to implement. A more significant advantage is its robustness to relative frequencies which vary over time. In fact, the instantaneous approach actually manages to avoid explicitly estimating the relative frequencies. In general, the more the frequencies vary of time, the poorer moving window MFQL performs. A third advantage to using the instantaneous estimates in MFQL is that it incorporates the interval since the last time a video was batched into its weighting factor. We are thus assured that starvation will never occur: If a video has not been scheduled for a long period of time, its priority will grow accordingly.

Having decided on instantaneous MFQL, we now compare its performance relative to the idealized MFQL scheme in which the relative frequencies are known a priori. We also implemented the instantaneous MFQL policy by employing the arrival lengths of the videos instead of the queue lengths. Recall that this priority policy effectively ignores user defection behavior. Figures 9 and 10 show average latency, while Figures 11 and 12 show defection probability and unfairness, respectively. Arrivals rates are

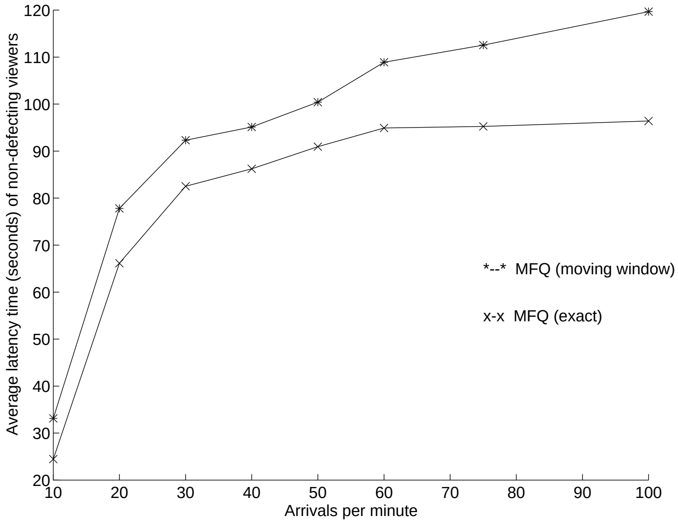

Figure 9: Average Latency Times for Varying Arrival Rates

varied in Figures 9 and 11, while server capacity is varied in Figures 10 and 12. Overall, it would appear that instantaneous MFQL has performance fairly close to the theoretical bound given by the idealized MFQL algorithm. Surprisingly, there does not seem to be a case for using arrival rather than queue lengths. In each metric performance of the arrival algorithm is close but generally inferior to that of the queue length algorithm.

6 Conclusions

In this paper we have presented a simple but powerful concept called the *factored queue length* as a basis for scheduling batched videos in a VOD environment. The factored queue length is derived from an analytical model which optimizes the average latency time of a viewer in the absence of defections. The purpose of presenting and solving this model was to provide intuition regarding a practical optimal policy for batching. This led to the proposed MFQL policy. We performed empirical simulations and found that the performance of the idealized MFQL policy is better than the other standard policies in terms of the average latency time if there are no defections. Given that defections do exist, the MFQL policy outperforms both the FCFS and MQL policies in terms of the defection probability, and it performs better than MQL and nearly comparably to FCFS in terms of fairness. The average latency time of the MFQL policy is much better than that of FCFS. Given that the performance of instantaneous MFQL approximates that of its idealized variant, the instantaneous MFQL policy appears to be an excellent choice for a batching algorithm. It exhibits excellent performance with respect to all of the standard performance measures.

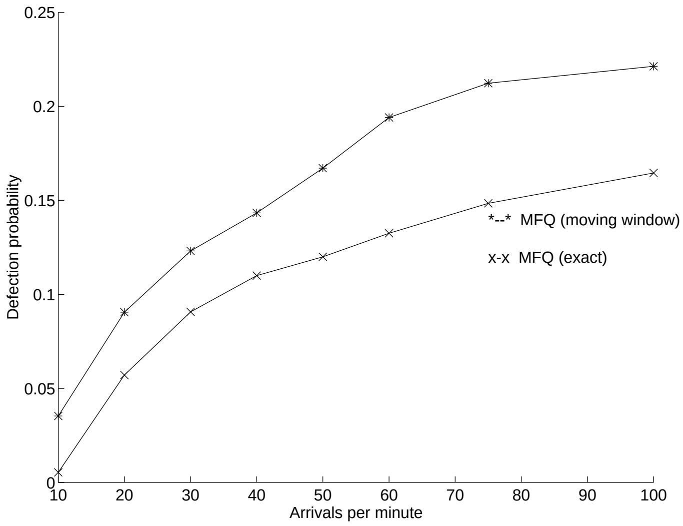

Figure 10: Average Latency Times for Varying Server Capacities

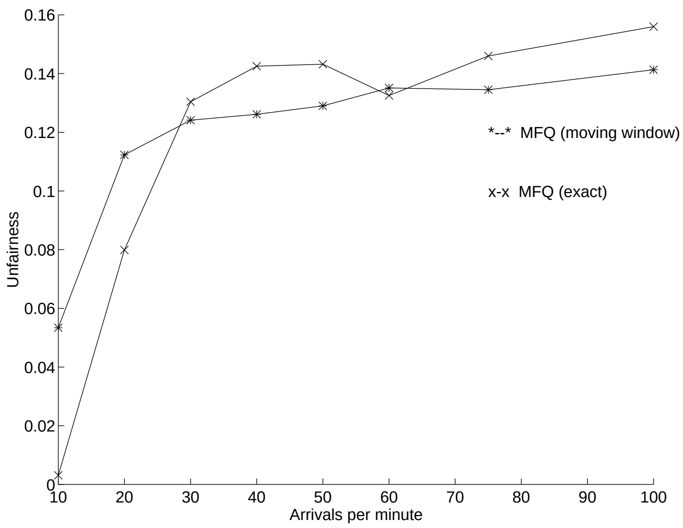

Figure 11: Defection Probabilities for Varying Arrival Rates

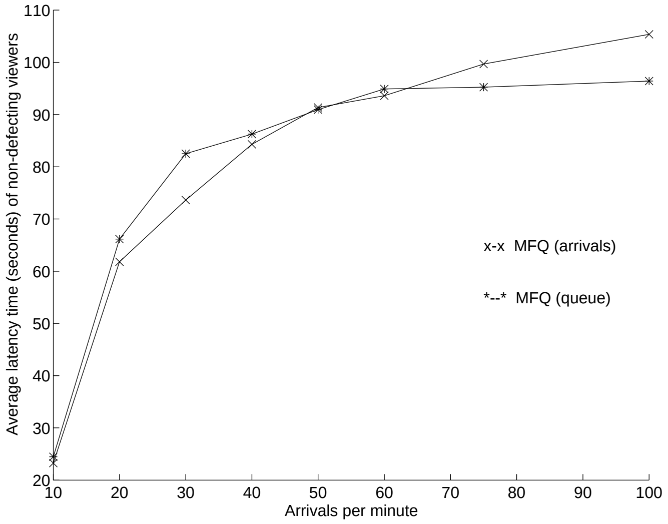

Figure 12: Unfairness for Varying Server Capacities

References

[1] Anderson, D. P., "Metascheduling for Continuous Media," *ACM transactions on Computer Systems,* Vol. 11, No. 3, 1993, pp. 226-252.

[2] Avriel, M., "Nonlinear Programming Analysis and Methods," Prentice Hall, Englewood Cliffs, NJ, 1976.

[3] Chen, M-S., Kandlur, D., and Yu, P., "Storage and Retrieval Methods to Support Fully Interactive Playout in a Disk-Array Based Video Server," *Multimedia Systems*, Vol. 3, No. 3, pp. 126-135, 1995.

[4] Dan, A., Sitaram, D., and Shahabuddin, P., "Scheduling Policies for an On-Demand Video Server with Batching," *Proceedings of the 2nd ACM Multimedia Conference*, San Francisco, CA, 1994, pp. 15-24.

[5] Dey-Sircar, J., Salehi, J., Kurose, J., and Towsley, D., "Providing VCR Capabilities in Large-Scale Video Servers," *Proceedings of the 2nd ACM Multimedia Conference*, San Francisco, CA, 1994, pp. 25-32.

[6] Gelman, A. D., and Halfin, S., "Analysis of Resource Sharing in Information Providing Services," *Proceedings of IEEE Global Telecommunications Conference and Exhibition 1990,* Vol. 1, 1990.

[7] Golubchik, L., Lui J. C. S., and Muntz, R., "Reducing I/O Demand in Video-On-Demand Storage Servers," *ACM Sigmetrics*, Ottawa, Canada, 1995, pp. 25-36.

[8] Golubchik, L., Lui J. C. S., and Muntz, R., "Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers", *Multimedia Systems*, Vol. 4, No. 3, 1996, pp. 140-155.

[9] Ibaraki, T., and Katoh, N., "Resource Allocation Problems: Algorithmic Approaches," MIT Press, Cambridge, MA, 1988.

[10] Kamath, M., Towsley D., and Ramamritham, K., "Buffer Management for Continuous Media Sharing in Multi-Media Database Systems," Technical Report 94-11, University of Massachusetts, 1994.

[11] Knuth, D., "The Art of Computer Programming," Vol. 3, Addison Wesley, Reading, MA, 1973.

[12] Pasquale, J. C., Polyzos, G. C., Anderson, E. W., and Kompella, V. P., "The Multimedia Multicast Channel," *Proceedings of the 3rd International Workshop on Network and Operating Systems,* 1992, pp. 197-208.

[13] Ozde, B., Bilris, A., Rastogi, R., and Silberschatz, A., "A Low-Cost Storage Server for Movie on Demand Databases," *Proceedings of the 20th Very Large Data Bases Conference,* Santiago, Chile, 1994.

[14] Rangan, V., Vin, H. M., and Ramanathan, S. "Designing an On-Demand Multimedia Service," *IEEE Communication Magazine,* Vol. 30, pp. 56-65, 1992.

[15] Shachnai, H., and Yu, P., "The Role of Wait Tolerance in Effective Batching: A Paradigm for Multimedia Scheduling Systems," Research Report RC 20038, IBM T J Watson Research Center, Yorktown Heights, NY, 1995.

[16] Tobagi, F., Pang, J., Baird, R., and Gang, M., "Streaming RAID - A Disk Array Management System for Video Files," *Proceedings of the 1st ACM Multimedia Conference*, Anaheim, CA, 1993, pp. 393-400.

[17] Yu, P., Chen, M-S., and Kandlur, D., "Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management," *Multimedia Systems*, Vol. 1, No. 3, pp. 99-109, 1993.

[18] Yu, P., Wolf, J., and Shachnai, H., "Design and Analysis of a Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications," *Multimedia Systems*, Vol. 3, No. 4, pp. 137-149, 1995.