Simulations on Batching in Video-on-Demand Transmissions*

Juan Segarra and Vicent Cholvi

Departament de Llenguatges i Sistemes Informàtics,
Universitat Jaume I, 12071 Castelló, Spain
{jflor,vcholvi}@lsi.uji.es

Abstract. One of the methods for taking advantage of multicast services is the use of batching. With this method, several request of the same video are grouped and transmitted together, using only the bandwidth required for one transmission. This method is commonly used in transmission of streamed data. In this paper we analyze the system performance with explicit constant batching, and demonstrate that a system without explicit batching performs better in terms of delays. We also propose a dynamic batching policy which improves the system performance both in mean and in maximum serving times.

1 Introduction

With the advancement of broadband network technology, Video-on-Demand (VoD) services are becoming commonplace, specially in local residential areas. A VoD system is typically implemented by a client-server architecture supported by certain transport networks. Among the many issues that have attracted the attention of researchers, optimizing the network channel demand is considered a major one. In order to solve that problem, VoD systems usually serve multiple clients at the same time by means of artificially delaying several requests for the same video [5,6,11,13] and serving them together. This method is known as *batching*.

However, these artificially introduced delays may not be always necessary or even convenient. For instance, in a system with low load, all video requests could be served immediately, and the use of batching would be clearly inefficient, since it only introduces an unnecessary delay. Moreover, the amount of bandwidth used in a high load period is much bigger than in low ones, and since these networks have to afford this requirement, in low load periods this resource is mostly underutilized despite of offering Internet connections or other similar services. Thus, bandwidth is not always a critical resource.

In this paper we focus on the convenience of using explicit batching. We will evaluate, by means of some simulations, how the use of batching improves (or reduces) the overall performance of the system described in Section 2. We

* This study is partially supported by the CICYT under grant TEL99-0582 and Bancaixa under grant P1-1B2000-12.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 453–462, 2003.

consider the *serving time* (the time between a video request and the beginning of its transmission) as the metric to evaluate the system performance.

First, we use a *Constant Explicit Batch-Time Policy* with different delays. Whereas this policy is the most widely used, we show that, in general, it provides improvements on bandwidth, but it does not provide any delay improvement against a system without explicit batching (but still with implicit batching). Then, we introduce a new batching policy which varies dynamically following the system load. Even though such a policy has a better performance in starting delays than previous ones, it only provides a low improvement when the system is highly saturated.

The rest of our paper is organized as follows. In Section 2 we describe our video-on-demand delivery system and the considered scenario. In Section 3 we show our comparison between constant batching policies and in Section 4 we present our approach and results with dynamic batching. Finally, our conclusions are presented in Section 5.

2 System

For our study, we use a quite simple network architecture consisting of a central server, storing all videos. Users are connected to that server by means of a common link which has a bandwidth of 1.5 Gbps downstream. A moderate bandwidth is also needed upstream in order to send user requests to the central server. We consider that this is a VoD dedicated network[1], therefore our goal is to obtain the best results using the available resources. We use a configuration consisting of 1 000 videos and 15 000 users. Each video information has been constructed making variations over the example presented in [12], resulting in videos of a mean duration of about 85 minutes and a mean bandwidth requirements of about 1 850 Kbps in fragments ranging from 96 to 4 608 Kbps. With this configuration we ensure that the link will become *saturated* (i.e. it will reach an utilization of 100%) at some period of time, being non-saturated during the remaining time. That will allow us to analyze the system behavior in both situations.

On the other hand, transmitting real-time VBR flows is not a trivial matter because a different amount of bandwidth will be needed during the transmission [3,8]. One of the methods to improve these transmissions is the *smoothing* [7, 9,12] of streams before their transmission. This way peaks and rate variability requirements are minimized. We use an approach similar to [12], which develops a *transmission plan* consisting of time periods so that, in each period, the transmission may be performed by using a constant-bit-rate (CBR) network service. Thus, each video is characterized by a collection of tuples $\langle time, rate \rangle$, whose first parameter denotes a time period and whose second one is the CBR rate associated with such a time period.

[1] Usually these networks offer more services, but dedicating a concrete amount of bandwidth to each service is also usual.

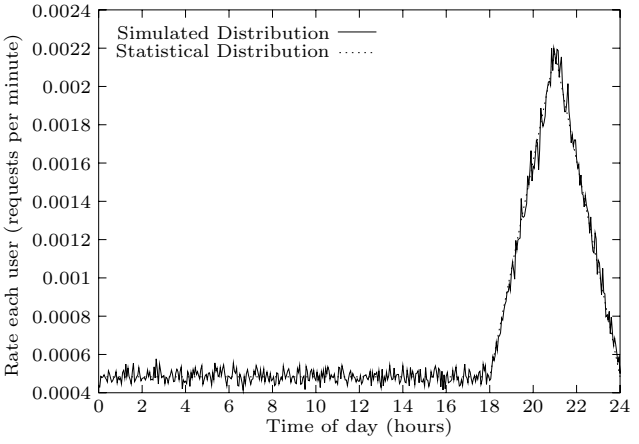

Fig. 1. Request rate distribution in a 24 hour period.

In our approach to the video transmission problem we study the performance in a 24 hour period. Previous studies on video demand rates [2,10] have determined the behavior of these rates in this period, and our simulations work with a rate distribution according to these studies. We also consider that a user makes a request only if he or she is not waiting nor receiving a video (i.e. nobody can receive more than one transmission at a time). In Fig. 1 we can see the statistical request rate and the request rate distribution obtained in a simulation having one request per day for each user. This is what we assume in our study (actual VoD trials show that the average number of user request per week is between 2 to 3).

Furthermore, each video has a *popularity*, which represents the probability of being requested according to Zipf's law [4]. This distribution has been found to statistically fit video program popularities estimated through observations from video store statistics [1].

In order to guarantee that, once a video is accepted (maybe after some start-up delay), it will be delivered without interruption to the final user(s), it is necessary to use a *reservation algorithm* to manage video requests. It works as follows: when a video request is received, we test (for each video-part) if it is already planned for transmission at the same time. If so, that request is added to the multicast transmission of this video-part without any additional bandwidth requirements. Otherwise, we check if there is enough bandwidth during the slots of the transmission. In that case, bandwidth is reserved by adding the new video-part rate to the bandwidth used. See Fig. 2 for details.

As it can be seen, this reservation algorithm guarantees that all requests will be served, maybe after some start-up delay. Moreover, users can know the exact time when the transmissions will begin, so we assume that there are no cancellations.

Step #1: A user requests a video.
Step #2: If that video has been already requested and it has not started yet, the start time of the new request will be the same as the previous one. Otherwise, the new request will start after an explicit batch-time.
Step #3: The system calculates the exact time when each video-part has to be transmitted.
Step #4: For each video-part do
If there is a reserve for this video-part at the same time, add our request as a multicast. Otherwise, reserve the bandwidth needed for this video-part during the required time slots.
Step #5: Accept video for transmission or go to Step #3 in the next time slot (in which case, all reserves and multicasts for this request performed in Step #4 are canceled).

Fig. 2. Algorithm for the acceptance of video requests.

An important feature of the reservation algorithm is the fact that, in saturation periods, all new request will have to wait to be served. So, even in the case where the there is not explicit batch-time, the reservation algorithm will still induce an *implicit batching* that, as we will show in the next sections, provides a very good system's performance.

3 Constant Batching Policies

Using a constant explicit batching scheme is currently the most used policy maybe because of its simplicity. Basically it consists of taking a constant batch-time delay throughout the whole system execution.

In this section, we analyze the performance of our system when taking explicit batch-time values of 0, 10, 20 and 30 minutes (despite we have tested more values, for clarity we only present results for these four values, since we found the rest follow the same pattern). Fig. 3 shows the starting delays in a 24 hour period.

First of all, it can be readily seen that, when the request rate is low (from approximately 0h to 19h), the more we increase the batch-time the more the starting delay increases. On the other hand, Fig. 3 also shows that during the time period when the request rate is high (from approximately time 19h to 24h) starting delays are all following the high audience peak independently of the batch-time used. This behavior can be explained if we look at the reservation algorithm in Fig. 2. This algorithm tries to allocate transmissions as soon as possible after their batch-time. However, in the time interval where the system is saturated, the delay necessary to start transmitting is higher than the batching delay. Consequently, since the system gets saturated in a short time interval (a few minutes), at the very end, all policies operate in the same way most of this time period, confirming what has been observed in Fig. 3.

In Fig. 4 we can see that, in the low load period, bandwidth usage never gets saturated. Therefore, since there is enough bandwidth to serve immediately all

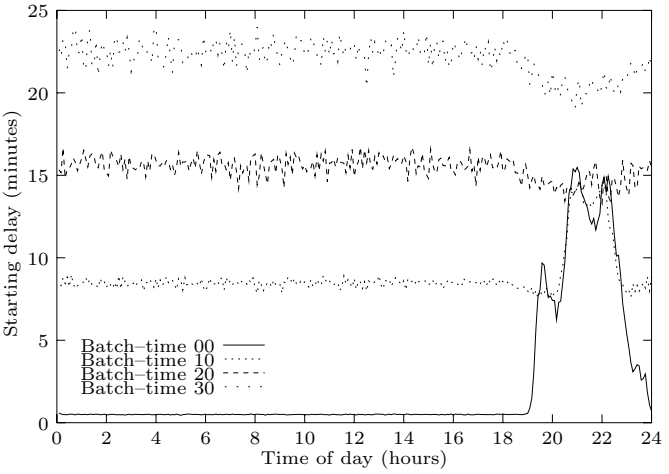

Fig. 3. Mean starting delays during a 24 hour period.

requests, adding a batch-time will only increase their starting delays. In turn, taking smaller batch-times will increase the used bandwidth. However, here we are assuming that the whole bandwidth is completely dedicated to our video-on-demand system. On the other hand, it can be seen that with very high explicit batch-times it is possible to avoid saturation. However, it also prevents from obtaining better starting delays.

4 Dynamic Batching

We have demonstrated, in the previous section, that a policy without explicit batching performs better in delay terms than with constant batch-times. In this section we present a dynamic batching approach to the problem. Such an approach consists of obtaining a batch-time that adjusts itself according to the current system load.

Our idea is the following. Think that the system receives a video request in the instant t, and the objective is to compensate the load we will have in the instant $t + \Delta t$. This compensation is done using batching over the requests in the current time t. The batch-time we assign to a request on time t is defined as follows:

$$batch_t = load_{t+\Delta t}\, adjust$$

where $load_{t+\Delta t}$ represents the number of videos awaiting for transmission in the time slot $t + \Delta t$ and $adjust$ is a function that will be used to adjust the load with the batch-time. In order to make load independent of the system, $load_{t+\Delta t}$ has been normalized so that a value of 1 indicates the saturation point.

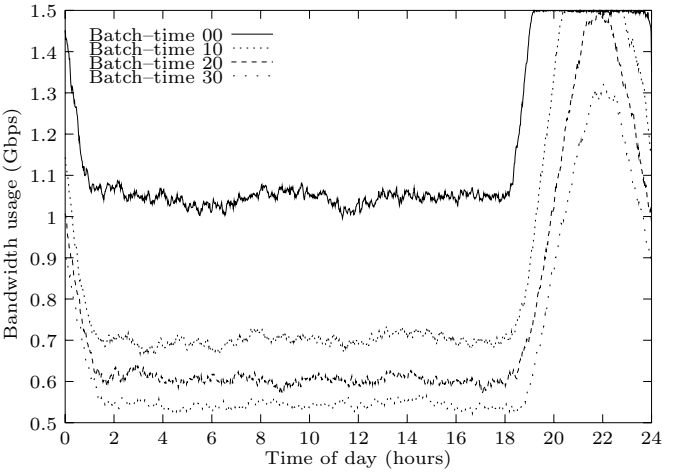

Fig. 4. Bandwidth usage during a 24 hour period.

To define *adjust*, think in a system which is continuously receiving requests and immediately serving those requests at the same rate. This system has, in addition, a list of video requests waiting for being served. Therefore, when a request for a given video is received it may happen that that video is also in the list waiting for being served. Thus, both requests would be transmitted as a multicast and the additional load list would decrease in one video. If this procedure is used until all waiting videos in the list are transmitted, at that time we would have eliminated the additional load list. Let us call $EliminationTime$ the time where the additional load is eliminated.

$$EliminationTime = AdditionalLoad\,\frac{1}{ProbCoincidence}$$

where $AdditionalLoad$ is the additional load list (normalized as above) we have to eliminate, and $ProbCoincidence$ is the probability of having a coincidence when requesting a video for transmission, that is, the probability of the requested video being in the load list.

For our work, we choose $batch_t \equiv EliminationTime$. Since we have that $load_{t+\Delta t} \equiv AdditionalLoad$, then $adjust \equiv \frac{1}{ProbCoincidence}$. Therefore,

$$batch_t = \frac{load_{t+\Delta t}}{ProbCoincidence}$$

A detailed description of $load_{t+\Delta t}$ and $ProbCoincidence$ can be found in the Appendix.

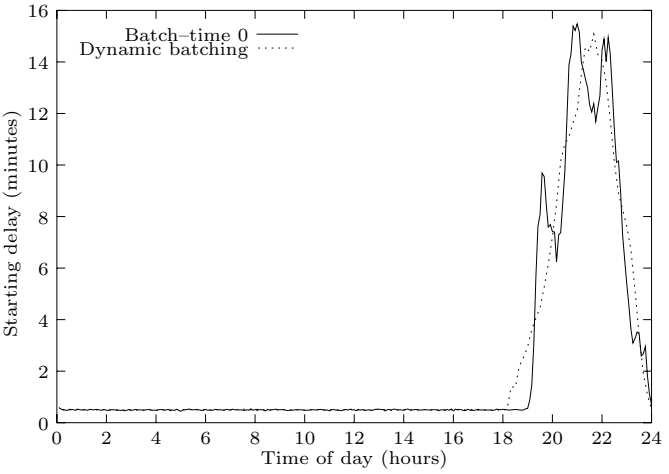

Fig. 5. Starting delays during a 24 hour period.

Results

Now, we show a comparison between this policy and one without explicit batching (which has been shown in the previous section to provide better results than using any explicit constant batching policy).

Fig. 5 shows the starting delays both without explicit batching and with a dynamic batching policy. During the low load period, both policies perform very similarly and between 0 and 1 minute. When the high load period begins, the dynamic policy responds adding artificial delays for grouping requests. This action produces a sooner increase in starting delays but this prevents starting delays from increasing so much as without explicit batch-time when these delays are higher. That is because the dynamic policy has a look ahead component, whereas the batch 0 policy has no information about the future system load.

The mean starting delay during a 10 days period with the dynamic policy in this case is 4 min 21 sec, and its maximum delay has been 27 min 22 sec. These values are about a 0.4% and a 4.7% lower than without explicit batch-time.

Bandwidth usage with these configurations is presented in Fig. 6. Whereas the bandwidth usage is almost the same, with the dynamic policy the added batching produces a delay in saturation. The reason is the same as above, grouped requests are transmitted together in multicast transmissions and they use less bandwidth.

5 Conclusions

In this paper we have studied the effect of using batching in the transmission of video on demand. Instead of using cost functions, this study presents results

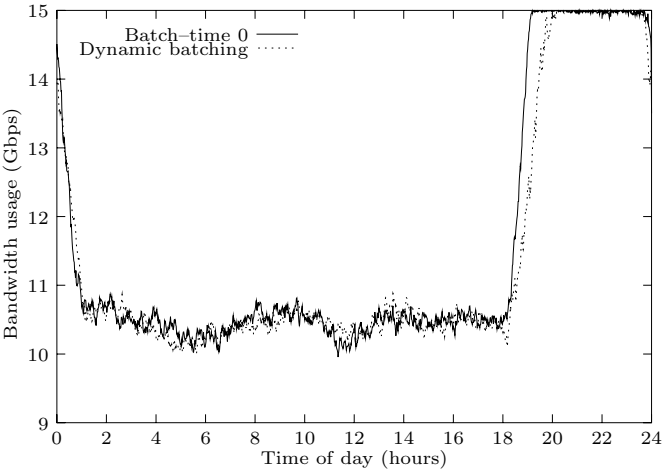

Fig. 6. Bandwidth usage during a 24 hour period.

based on the delays between video requests and the beginning of their transmission. Firstly, we have demonstrated that, without using explicit batching techniques, the overall system performs better than using a constant explicit batch-time. Furthermore, to improve the system performance, we have proposed a method for statistically calculate the system load, and use the resulting value to adjust the batch time dynamically. With this method we have reduced around 0.4% the mean starting delay and around 4.7% the maximum starting delay compared with a system without batching.

These results demonstrate that, unless a constant explicit batch-time is needed for some specific reason, it is better not using it. In case of needing a more optimized system, we also demonstrate that our dynamic batching method is better than using a batch-time 0 policy. However, the improvements are relatively small and require additional computation, so a policy without explicit batching could also be considered as a good option due to it performs well and it is the simplest batching policy. The reasons of the good performance are that without explicit batching there is no artificial delay introduced in the system, and this policy has a good adaptation to the current load, since it allocates the transmissions as soon as possible.

References

1. A. Dan and D. Sitaram and D. Shahabuddin. Dynamic Batching Policies for an On–Demand Video Server. *Multimedia Systems*, 4:112–121, 1996.
2. Bell Atlantic. Fact Sheet: Results of Bell Atlantic Video Services. Video–On–Demand Market Trial. Trial Results, 1996.

3. L. Berc, W. Fenner, R. Frederick, and S. McCanne. Rpt payload format for jpeg-compressed video. Request for Comments 2035, Network Working Group, October 1996.
4. Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM '99 conference*, March 1999.
5. Shueng-Han Gary Chan and Fouad Tobagi. Providing on-demand video services using request batching. *IEEE Int. Conf. Communications (ICC'98)*, pages 1716–1722, June 1998.
6. Asit Dan, Dinkar Sitaram, and Perwez Schahabuddin. Scheduling policies for an on-demand video server with batching. *ACM Multimedia*, pages 15–23, 1994.
7. W. Feng, F. Jahanian, and S. Sechrest. An optimal bandwidth allocation strategy for the delivery of compressed prerecorded video. *Multimedia Systems*, 5(5):297–309, 1997.
8. D. Gall. Mpeg: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.
9. Jeniffer Rexford and Don Towsley. Smoothing Variable–Bit–Rate Video in an Internetwork. *IEEE/ACM Transactions on Networking*, pages 202–215, April 1999.
10. PG de Haar et al. DIAMOND Project: Video–on–Demand System and Trials. Eur Trans Teleccommun (8)4: 337–244, 1997.
11. Juan Segarra and Vicent Cholvi. Distribution of video–on–demand in residential networks. 8^{th} *International Workshop on Interactive Distributed Multimedia Systems*, September 2001. Lecture Notes in Computer Science 2180.
12. Arun Solleti and Kenneth J. Christensen. Efficient transmission of stored video for improved management of network bandwidth. *International journal of network management*, 10:277–288, 2000.
13. Constantinos Vassilakis, Michael Paterakis, and Peter Triantafillou. Video placement and configuration of distributed video servers on cable TV networks. *Multimedia Systems*, 8:92–104, 2000.

Appendix: Description of Dynamic Batching Parameters

Definition 1 *We define the* real load *in a time slot t as:*

$$rload_t = \frac{|(Awaiting_{t-1} \cup Requested_t) - Served_t|}{ServingCapacity}$$

where $Served_t$, $Requested_t$ and $Awaiting_t$ respectively denote the set of different videos served, requested and awaiting for transmission in the time slot t, and $ServingCapacity$ is the mean number of different videos served in a time slot using all bandwidth.

Nevertheless, this definition is not usable in practice, because it needs both which videos are awaiting for transmission and which of them will be served in each time slot. Obviously, it is not known in advance which videos will be requested, nor which ones will serve the system since that depends on the batch-time assigned to each request. Therefore, we need to obtain a (statistical) load value independent of the batching algorithm. All probability operations below are done using the Zipf's distribution, which offers probability values based on popularity.

Definition 2 *We define the* statistical load *in a time slot t as:*

$$sload_t = max\left(0, sload_{t-1} + \frac{NewRequests_t - ServingCapacity}{ServingCapacity}\right)$$

In this equation, we calculate the load in a time slot t by adding the load generated by the new requests to the load of the previous time slot and subtracting the amount of load transmitted, which is 1 because of the normalization[2].

The $NewRequests_t$ parameter is the number of new requests in the time slot t, and it is calculated as:

$$NewRequests_t = ProbNewReq(ReqInSlot(t), sload_{t-1}\, ServingCapacity)\, ReqInSlot(t)$$

where $ReqInSlot(t)$ is the number of requests in the time slot t obtained statistically using the request distribution and $ProbNewReq(new, awaiting)$ is the probability of that *new* requests not being included in *awaiting* requests. Statistically $ProbNewReq(new, awaiting)$ is calculated as $P(Videos(new) \mid Videos(new) \nsubseteq Videos(awaiting))$, being $Videos(x)$ a set of x $VideoID$s obtained statistically from the repository using Zipf's law.

However, obtaining this statistical load value has a computation cost of $O(MaxVideos^{new})$ in the step $ProbNewReq(new, awaiting)$, so an approximation is needed. We obtain this approximation supposing that the proportion of different requests in each time slot is a constant k, and the $sload_{t-1}$ videos awaiting for transmission are the most popular ones. The first simplification is reasonable when the number of requests in each time slot is not very different and in our case it is less than one order of magnitude. The second one is reasonable having in mind that Zipf's law popularity implies exponential probabilities of request, so the most popular videos are much more requested than the others. The usable $ProbNewReq(new, awaiting)$ definition would be the probability of not requesting the *awaiting* most popular videos. Statistically: $k\, P(x \mid x > awaiting)$, where k is the proportion of distinct elements in a group of $VideoID$s requested in each time slot.

The remaining definition is $ProbCoincidence$, but it is exactly the opposite of $ProbNewReq()$, so its calculation is immediate.

The simulation parameters can be easily obtained. $ServingCapacity$ is obtained dividing the link capacity by the product of the mean bandwidth usage of all videos each minute and the mean video duration. The resulting value was 9.42 videos. The parameter k is obtained using the mean value from a complete simulation. The resulting value was $k = 0.77$.

To obtain the best value of Δt, we have simulated our system with Δt ranging from 0 to 10 hours. Depending on the pursued objective this value can be adjusted closely, in our case we take a value of 60 minutes, which offers good results both for mean delay and maximum delay.

[2] Actually, we do not know the value representing the transmitted videos, so we assume the system is serving at full capacity (1 because of the normalization) and then use the *max* operation to prevent the system from getting load values lower than 0.