

Catching and Selective Catching: Efficient Latency Reduction Techniques for Delivering Continuous Multimedia Streams^{*}

Lixin Gao[†] Zhi-Li Zhang[‡] Don Towsley^{*}

Abstract

We present a novel video streaming technique called catching for on-demand delivery of “hot” (i.e., frequently accessed) video objects to a large number of clients. This technique not only significantly reduces the server and network resource requirements but also is capable of providing near-instantaneous service to a large number of clients. We prove that the performance of catching is close to the best achievable by any broadcasting scheme that supplies near-instantaneous service. By combining this technique for delivery of “hot” video objects with controlled multicast [8] for delivery of “cold” video objects, we design an efficient video delivery scheme referred to as selective catching. Extending this scheme to a proxy-assisted video delivery environment, we also develop a proxy-assisted selective catching scheme. Through empirical studies, we demonstrate the efficacy of the proposed video delivery schemes.

1 Introduction

The past few years have seen the dramatic growth of multimedia applications which involve video streaming over the Internet. Server and network resources (in particular, server I/O bandwidth and network bandwidth) have proved to be a major limiting factor in the widespread usage of video streaming over the Internet. In order to support a large population of clients, techniques that can efficiently utilize server and network resources are essential. In designing such techniques, another important factor that must be taken into consideration is the *service latency*, i.e., the time a client has to wait until the object he/she

[†] Department of Computer Science, Smith College, Northampton, MA 01060, USA. gao@cs.smith.edu

[‡] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN. 55455, USA. zhzhang@cs.umn.edu

^{*} Department of Computer Science, University of Massachusetts, Amherst, MA 01030, USA. towsley@cs.umass.edu

^{*} The first author was supported in part by NSF grant NCR-9729084, and NSF CAREER Award grant ANI-9875513, and the second author was supported in part by NSF CAREER Award grant NCR-9734428. The third author was supported in part by NSF grant ANI-9805185. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

has requested is started to playback. The effectiveness of a video delivery technique must be evaluated in terms of both the server and network resources required for delivering a video object and the expected service latency experienced by clients. Clearly, “popularity” or access pattern of video objects (i.e., how frequent a video object is accessed in a given time period) plays an important role in determining the effectiveness of a video delivery technique.

In this paper we propose and develop a novel video delivery technique called *catching*, which can efficiently utilize server and network resources while providing near instantaneous service to clients. This technique is particularly suitable for “hot” (i.e., frequently access) video objects. The effectiveness of the technique is achieved through intelligent integration of the “server-push” and “client-pull” video delivery paradigms. Using this technique, a video server periodically “broadcasts” a video object via a number of *dedicated multicast channels*. A client who wishes to watch the video immediately joins an appropriate multicast channel without waiting for the beginning of the next broadcast period. At the same time, the client sends a request to the server to retrieve the missing initial video data (referred to as the *prefix* of the video object). The prefix is delivered by the server using a unicast channel and played back immediately by the client. On the other hand, the video data received from the multicast channels will be temporarily buffered at the client until they are played back. Hence, catching provides the minimal service latency by allowing a client to join the on-going multicast channels to receive video data “pushed” by the server while “pulling” the missing video data from the server via unicast channel. Using a smart broadcast scheme such as the *Greedy Disk-conserving Broadcast* (GDB) scheme [7], we can minimize the *expected* server and network channels needed to deliver a video object, given its access pattern. This is verified through simulations. Furthermore, we prove that the number of channels required by catching is close to the minimum achievable by any broadcasting scheme that supplies near-instantaneous service.

In order to account for the diverse access patterns for a collection of video objects in a video server, we design an efficient video delivery scheme called *selective catching* which combines catching with another video delivery technique – *controlled multicast*. Controlled multicast is a “client-pull” technique which is most effective in delivering “cold” video objects. Based on video access patterns, we introduce a simple policy for classifying “hot” and “cold” video objects and apply catching and controlled multicast accordingly to deliver the video objects to clients. Through empirical studies, we demonstrate that in terms of both server/network resource requirements and service latency, *selective catching* outperforms either catching or controlled multicast applied alone.

The proposed catching and selective catching techniques can also be applied in a proxy-assisted video delivery environment [12, 13] where initial partial video data (prefixes) of some video objects can be staged in proxy servers in a pre-determined manner. Under this proxy-assisted video delivery architecture, we can take advantage of the resources (processing and disk storage) available at proxy servers to significantly reduce the server and (backbone wide-area) network resource requirements while at the same time providing instantaneous or near-instantaneous service to clients. We present a brief description of the algorithms used by a central video server, proxy servers and clients to coordinate the video delivery using proxy servers. Simulations are carried out to illustrate the significant reduction in server and network resource requirements achieved using this proxy-assisted video delivery architecture.

The remainder of this paper is organized as follows. The related work is briefly surveyed in Section 1.1. In Section 2 we describe the problem setting and the background material necessary for the catching technique which we present in Section 3. The selective catching video delivery scheme is in-

roduced in Section 4. In Section 5 we apply the selective catching technique to the proxy-assisted video delivery architecture. The paper is concluded in Section 6.

1.1 Related Work

In recent years a variety of “client-pull” and “server-push” techniques for video delivery have been proposed (see, e.g., [6, 1, 3, 4, 7]). The simplest “client-pull” technique is to deliver a separate video stream upon each client request. This technique, while providing minimal service latency to a client, is obviously not efficient in terms of server and network resource utilization. Clever “client-pull” techniques such as *batching* [1, 6] and *patching* [5, 8] have been proposed that take advantage of the underlying network multicasting capabilities to reduce server and network resource requirements. In the case of batching, this reduction in server and network resource requirements is achieved through increased service latency, as it delays earlier requests for a video object until a certain number of requests for the same object arrive before the video object is scheduled to be delivered. Hence, batching is less effective for “cold” video objects. On the other hand, “patching”, which allows multiple clients to share a multicast channel whenever possible, is most effective in reducing the server and network resource requirements for “cold” video objects.

“Server-push” techniques are typically designed for “hot” video objects. They employ a fixed number of multicast channels to periodically broadcast video objects to a group of subscribers. The difference between various “server-push” techniques lies in the broadcast schemes used. These broadcast schemes determine the server and network resources required for broadcasting a video object. “Server-push” techniques have the advantage that they utilize server and network resources more efficiently. But this efficiency is achieved through increased service latency, as a client can only start receiving a video object at the beginning of next broadcast period.

The catching technique we propose reduces service latency while taking advantage of the efficiency of periodic broadcast schemes in utilizing server and network resources. It thus eliminates the shortcoming associated with periodic-broadcast-based “server-push” techniques. Catching is similar in spirit to the split and merge (SAM) protocol [14] proposed for interactive VOD systems, where a unicast stream is scheduled on demand by a client’s request. The selective catching technique further improves the overall performance by combining catching and controlled multicast to account for diverse user access patterns.

The problem of delivering continuous media streams using proxy servers has been studied in a number of contexts. In [12], we develop video staging techniques to store a pre-determined amount of video streams in strategically placed proxy servers to reduce the backbone network bandwidth requirement for delivering video streams across a wide-area network. In [13], a prefix caching scheme is proposed to reduce the latency while delivering smoothed variable-bit-rate (VBR) continuous streams between the proxy and clients. Proxy-assisted video delivery is also proposed in the context of the dynamic skyscraper delivery scheme in [10].

Our proxy-assisted catching scheme can improve service latency as well as reduce server and network resource requirements. Unlike [10], our scheme can handle variable object sizes, and is based on formal analysis of multicast scheduling policies. From this analysis, the design parameters can be derived in a straightforward manner. As a result, our solution can be optimized accordingly.

2 Problem Overview and Preliminaries

2.1 Problem Overview

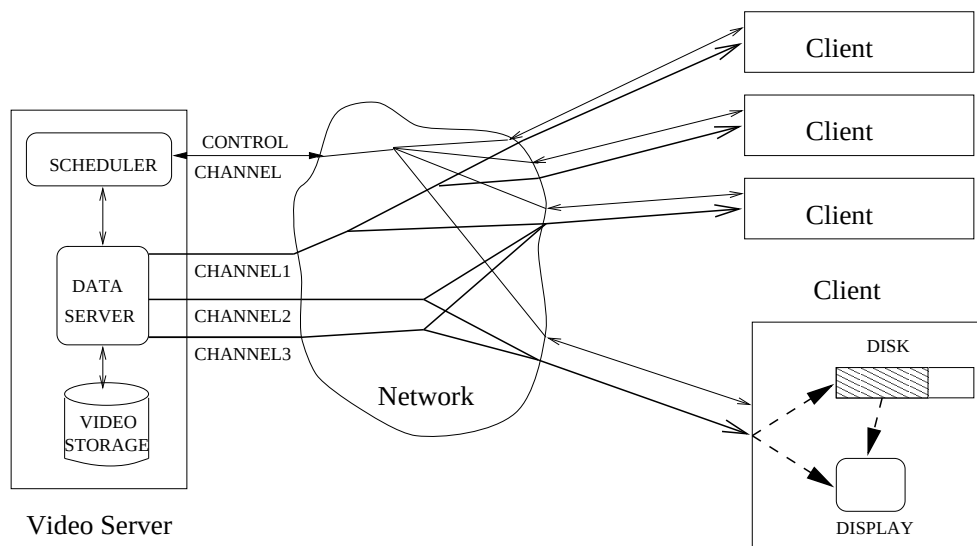


Figure 1: An overview of VOD system architecture.

Consider a typical stored video-on-demand (VOD) delivery system as shown in Figure 1. A central video server delivers video streams from a video object store to a large number of clients across a high-speed network. The central server organizes the server and network resources required to deliver a video stream¹ into a (*logical*) channel. A channel can be either a *unicast* channel or a *multicast* channel. The server uses a unicast channel to deliver a video stream to a single client, whereas the server uses a multicast channel to deliver a video stream *simultaneously* to a group of clients (this group of clients is referred to as a *multicast group*). In addition to the logical channels used for delivering video streams (i.e., *video delivery channels*), we also assume that there are *control* channels to deliver signaling messages to a client or a group of clients and vice versa for control purposes (e.g., which video object is requested by a client, which data channels a client should tune in to, when to start video play-back, etc.). The video server has a scheduler which receives client requests for video objects via control channels, processes client request and determine when and which video delivery channels to deliver requested video objects to clients.

Each client contains a set-top box, a disk, and a display monitor. A client is connected to the network via a set-top box, which selects one or more network channels to receive a requested video object according to the instructions from the server. The received video data are either sent to the display monitor for immediate playback, or temporarily stored on the disk which is retrieved and played back on the display monitor later. The *client storage space* is the maximum disk space required throughout the client playback period. For ease of exposition, in this paper we assume that the client disk space is sufficiently

¹In this paper we use the term *video stream* to denote a continuous flow or “stream” of video data (belonging to a certain video object) delivered from the server to a client or a group of clients. As will be clear later, a single video object can be partitioned into segments and delivered using multiple video streams via several delivery channels.

large to store at least half a video². The *client network bandwidth* is the maximum client network bandwidth required to receive video data from the network throughout the client playback period. In this paper, we assume that client has the capability of receiving video data from two channels at the same time³.

We assume that there is a total of C logical channels available, and there are N video objects in the server object store. The length of the i th object, $i = 1, 2, \dots, N$, is L_i minutes long. We assume that the requests for the i th video object arrive according to a Poisson distribution with an expected inter-arrival time of $1/\lambda_i$, where λ_i is the request rate of video i .

Given a client request for a video object, the *service latency* experienced by a client is the amount of time that the client has to wait until he/she can start the playback of the requested video object. A key issue in the design of video delivery techniques is how to efficiently utilize server and network resources (i.e., use the least number of video delivery channels necessary for delivering a video object) while keeping the (expected) service latency experienced by clients as small as possible.

2.2 Preliminaries

We briefly describe two video delivery techniques we have developed earlier to provide the necessary background for the catching technique we will introduce in Section 3.

2.2.1 Controlled Multicast: An Optimal Patching

Controlled multicast is a patching technique which allows multiple clients to share a multicast channel without delaying a client request. As a result, it is capable of supporting a large number of clients while providing *near instantaneous service*. Controlled multicast differs from the patching technique proposed in [5] in that it employs a patching threshold to optimize the expected channels needed to deliver a given video object.

As a “client-pull” technique, controlled multicast allocates channels at the request of clients. The following example illustrates how control multicast works. Consider two requests spaced 5 minutes apart for a 90 minute long video. A *multicast* channel is allocated to transmit the entire video in order to satisfy the first request. The second request is satisfied by allocating a separate *unicast* channel to “patch-up”, i.e., transmit the first five minutes of the video to the client while in the same time requiring the client to prefetch the rest of the video from the first channel. Because the second client is five minutes behind, it will buffer continually five minutes of the video data.

Controlled multicast only allows clients to share a multicast channel to receive a video stream *when the later client requests for the same video object arrive within a certain time from the first client request*. Otherwise, a complete video transmission for the video object is scheduled using a new multicast channel. In other words, for each video object i , a threshold T_i is defined to control the frequency at which a

²This assumption is not essential, since our proposed schemes can be easily extended to a general case where clients have any given amount of disk storage space, as we will point out in Section 3. In all of our empirical studies, the amount of client disk storage space used is actually only at most one third of a video object.

³With the advent of high-speed access technologies such as ADSL and cable modems, this is not an unreasonable assumption.

complete stream of video i is delivered [8]. For simplicity of exposition, we describe controlled multicast assuming that there are infinite number of server channels. When the first request for video i arrives at time t , the scheduler schedules a complete video stream of video i on a channel, say MG_i , immediately. Any subsequent request for video i piggybacks from channel MG_i so long as the request arrives within T_i minutes from the starting time of the previous complete transmission of video i (which is time t in this case). Otherwise, the request is served by initiating a new complete transmission of video i . This process repeats forever. For the details of interaction between the server and clients, readers are referred to [8].

In [8], the optimal value for the threshold T_i of video i is derived and is given below

$$T_i = (\sqrt{2L_i\lambda_i + 1} - 1)/\lambda_i. \quad (1)$$

Using this optimal threshold, the expected server bandwidth required to deliver video i to clients is

$$\sqrt{2L_i\lambda_i + 1} - 1. \quad (2)$$

2.2.2 GDB: An Efficient Periodic Broadcast

Greedy Disk-conserving Broadcast (GDB) [7] is a “server-push” video delivery technique based on the innovative periodic broadcast schemes developed recently [1, 3, 4]. Under these schemes, a video object is partitioned into segments and each segment is periodically broadcasted via a dedicated channel. The sizes of these segments are carefully designed in such a manner that clients who wish to receive the video object can join the appropriate channels to receive various segments at *scheduled* times to ensure continuous playback of the video object. Figure 2 illustrates how the schemes work through a simple example. A video is partitioned into four segments: A, B, C, D. Each segment is broadcast periodically via a dedicated channel. Clients prefetch video data according to a schedule that ensures the continuous playback. Furthermore, clients are guaranteed a maximum service latency of d minutes with only 4 dedicated channels, where periodically broadcasting the video stream every d minutes would require 11 dedicated channels to guarantee the same maximum service latency. In effect, a complete stream is multicast every d minutes but using only 4 channels. The key issue in periodic broadcast schemes is to determine how to partition video objects into segments so as to enable continuous playback at clients. A method for partitioning video objects is referred to as a *partition function*, which determines the performance of a periodic broadcast scheme. Given K_i dedicated multicast channels, a partition function $f(n)$ divides a video object into K_i segments, $T_1, T_2, \dots, T_{K_i}$, as follows. For $n = 1, 2, \dots, K_i$, segment T_n contains $f(n)L_i / \sum_{m=1}^{K_i} f(m)$ minutes of video data, and its data starts at the $\sum_{m=1}^{n-1} f(m)L_i / \sum_{m=1}^{K_i} f(m)$ th minute of the video and ends at the $\sum_{m=1}^n f(m)L_i / \sum_{m=1}^{K_i} f(m)$ th minute of the video. In [7], a set of constraints on partition functions are derived based on resource availability at the client side. In particular, GDB is shown to be most efficient among all the existing schemes, given the same client resource constraints. For example, if the network bandwidth at the client side is only sufficient to support two channels (i.e., receiving from two channels simultaneously), the optimal partition function $f(n)$ used in

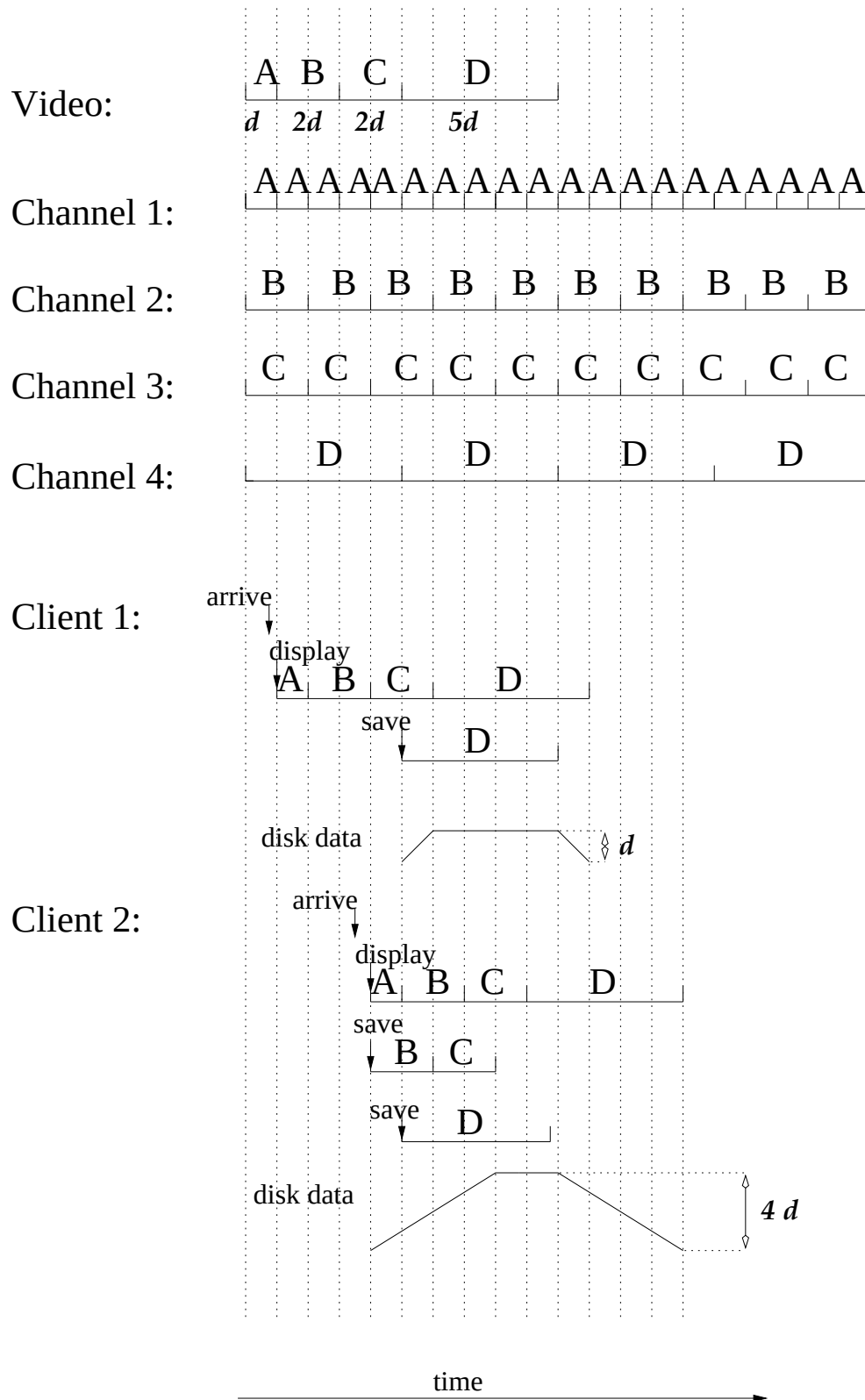


Figure 2: An example of periodic broadcast scheme.

GDB has the following form:

$$f_{GDB}(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2, & \text{if } n = 2, 3 \\ 5, & \text{if } n = 4, 5 \\ 12, & \text{if } n = 6, 7 \\ 5f(n-4), & \text{if } n > 7 \end{cases} \quad (3)$$

Given this partition function, it can be shown that the disk storage requirement at the client is equal to $(f_{GDB}(K_i) - 1)$ times the first segment size for a video object.

3 Catching

Periodic broadcast schemes can significantly reduce the number of channels required for delivering “hot” (i.e., frequently accessed) videos to large number of clients. However, they suffer from the shortcoming that clients experience increased service latency. In this section we present a novel video delivery technique — *catching*, which achieves the resource efficiency of periodic broadcast schemes while providing near instantaneous service to clients. This technique synergically combines the “server-push” and “client-pull” video delivery paradigms: Like periodic broadcast, catching dedicates a certain number of channels for periodic broadcasting. But unlike periodic broadcast, it reduces the service latency by allowing clients to join an on-going broadcast cycle at any time instead of waiting for the next broadcast cycle. Clients catch up with the current broadcast cycle by retrieving the missing initial frames (or a prefix) of the video object from the server via a separate unicast channel. Clients play back the prefix as it is received from the server, while at the same time temporarily storing the video data received from the broadcast channel into the disk. We illustrate how catching works using a simple example. As shown in Figure 3, a video object is partitioned into four segments, A, B, C and D, where A, B and C are of equal length, whereas D is twice as long. Therefore the server dedicates four channels to broadcast the four segments separately. Consider client 1 who arrives t seconds after the beginning of the current broadcast cycle of segment A. It joins the on-going broadcast cycle of segment A to receive the remaining data of segment A. At the same time it initiates a unicast “catch-up channel” via which the server delivers the first t seconds of video data of segment A to client 1. This “catch-up” stream is played back immediately by client 1, while the broadcasted data of segment A is temporarily stored. At the end of the current broadcast cycle of segment A, client 1 starts receiving segment B by joining the next broadcast cycle of segment B, while continuing the playback of segment A. At the end of the broadcast cycle of segment B, client 1 joins both the next broadcast cycles of segments C and D. By temporarily stored video data that do not need to be played back immediately, client 1 ensures the continuous playback of the video object. The behavior of client 2 is similar. The only difference is that at the end of the broadcast cycle of segment B, client 2 only needs to join the next broadcast cycle of segment C. As a result, client 2 needs to buffer at most s seconds of video data at any time, whereas client 1 needs to buffer up to $t + d$ seconds of video data. In both cases, client 1 and client 2 receive video data from at most two channels at any given time.

From the above example, we see that the partition function used in catching for periodic broadcasting can be derived directly from that used in a pure periodic broadcast scheme such as GDB. For example,

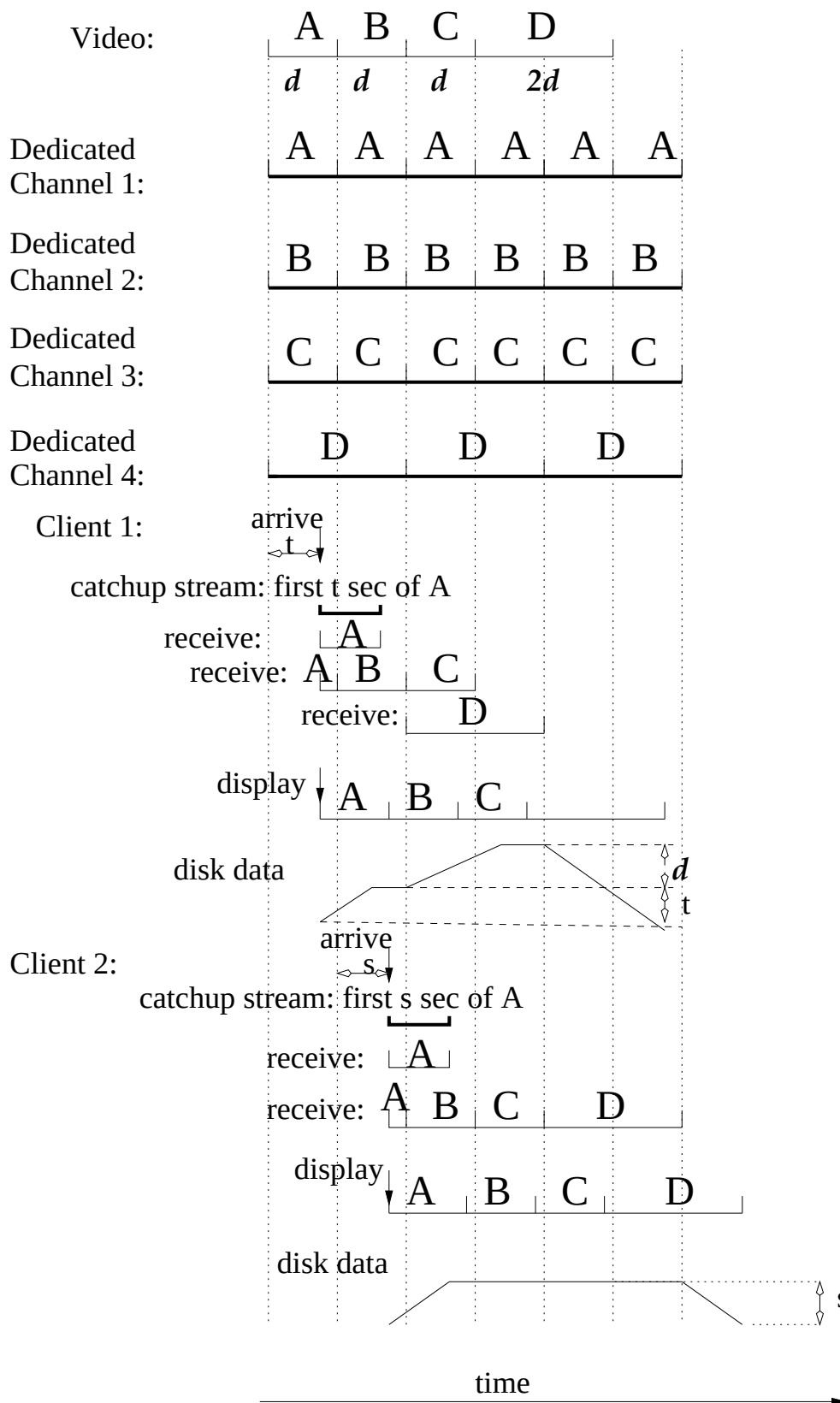


Figure 3: Illustration of catching.

given that the network bandwidth on the client side is only sufficient to support *two* channels, the partition function for catching is given by

$$f(n) = \begin{cases} 1, & \text{if } n \leq 3 \\ 2, & \text{if } n = 4, 5 \\ 5, & \text{if } n = 6, 7 \\ 12, & \text{if } n = 8, 9 \\ 5f(n-4), & \text{if } n > 9. \end{cases} \quad (4)$$

Comparing (3) and (4), we see that $f(n)$ is derived from f_{GDB} by adding two initial segments whose length is the same as that of the first segment of f_{GDB} . In other words, $f(1) = f(2) = f_{GDB}(1)$ and $f(n) = f_{GDB}(n-2)$ for $n \geq 3$. The first two identical segments are added to ensure that a client needs to join only one broadcast channel while receiving the “catch-up” stream from the server. Note that no matter when a client arrives during a broadcast cycle of the first segment, by the end of the broadcast cycle of the second segment, the “catch-up” stream of the first segment must have completely received and played back. Hence, after the end of broadcast cycle of the second segment, a client only needs to join the broadcast channels to receive the appropriate video segments. The client schedule for determining which channels to join and when to join is exactly the same as used in GDB. The only difference is that in catching clients always needs to buffer at least t seconds of video data at any time, if it arrives t seconds later than the beginning of an on-going broadcast cycle of the first segment.

Clearly, the ability of catching to achieve near-instantaneous service lies in the fact that, besides the dedicated broadcast channels, catching allocates additional unicast channels on-demand to deliver “catch-up” prefix streams. Hence the performance of catching is determined by the number of dedicated channels used for periodic broadcast as well by the number of “catch-up” channels needed. The rest of this section is devoted to the analysis of the *optimal* performance of catching for a video object with a known user access pattern. For simplicity of exposition, we will analyze the performance of catching based on the partition function $f(n)$ defined in (4). Namely, we assume that the network bandwidth on the client side is only sufficient to support two channels at the same time, and that clients have sufficient disk storage space to buffer at least half of the length of a video object in question. At the end of this section, we will briefly discuss how the analysis can be extended to more general cases.

Consider video object i whose length is L_i . Given the partition function $f(n)$, suppose K_i channels are dedicated to broadcast video object i and let F_i denote the length of the first broadcast segment of video object i . Then from the definition of the partition function, we have

$$L_i = F_i \sum_{n=1}^{K_i} f(n). \quad (5)$$

Under the assumption that client requests for video object i arrive according to Poisson distribution with an average rate λ_i , the *expected* number of unicast channels needed to deliver “catch-up” streams to clients is

$$\lambda_i F_i / 2. \quad (6)$$

This is because the expected length of the “catch-up” streams is $F_i / 2$. Hence, the total *expected* number of channels required for delivering video object i to clients is

$$K_i + F_i \lambda_i / 2. \quad (7)$$

From (5), we see that the smaller the number of dedicated broadcast channels K_i is, the larger the first broadcast segment F_i becomes. On the other hand, from (6), it is clear that the larger F_i results in more unicast channels are needed to deliver “catch-up” streams to clients. Therefore, there is trade-off between the number of dedicated broadcast channels and the expected number of unicast channels required for catching up. In order to optimize resource efficiency of catching, we minimize the total expected number of channels required to deliver each video object. This leads to the following optimization problem:

$$\begin{aligned} & \text{Minimize } K_i + F_i \lambda_i / 2 \\ & \text{subject to } L_i = F_i \sum_{n=1}^{K_i} f(n). \end{aligned}$$

Let K_i^* be the solution to the above optimization problem. Namely, K_i^* is the optimal number of dedicated broadcast channels such that the total expected number of channels required for delivery of video object i is minimized, i.e.,

$$K_i^* = \operatorname{argmin}_{K_i} (K_i + F_i \lambda_i / 2). \quad (8)$$

Therefore, the expected number of channels required for catching is

$$K_i^* + L_i \lambda_i / (2 \sum_{j=1}^{K_i^*} f(j)). \quad (9)$$

For the partition function $f(n)$ given in (4), K_i^* can be derived analytically, the detail of which is relegated to Appendix A. Here we provide an order estimate for K_i^* . From (4), it is not too hard to verify that $\sum_{j=1}^{K_i} f(j) = O(5^{K_i/4})$. Substituting this into (9) and taking the first-order derivative, we have

$$K_i^* = O(\log(\lambda_i L_i)). \quad (10)$$

Furthermore, the total expected number of channels required for video i is

$$\begin{aligned} & O(\log(\lambda_i L_i)) \\ & O(\log(\lambda_i L_i)). \end{aligned} \quad (11)$$

since the expected number of unicast “catch-up” channels required (see Equation (9))

$$L_i \lambda_i / (2 \sum_{j=1}^{K_i^*} f(j)) = O(1)$$

with optimal K_i^* . Comparing with (2), it is clear that for λ_i reasonably large, the total expected number of channels required by catching to deliver a video object is considerably less than that required by controlled multicast. In other words, for “hot” video objects, catching is much more efficient than controlled multicast. To verify this observation, let us consider an numerical example. In Figure 4 we plot

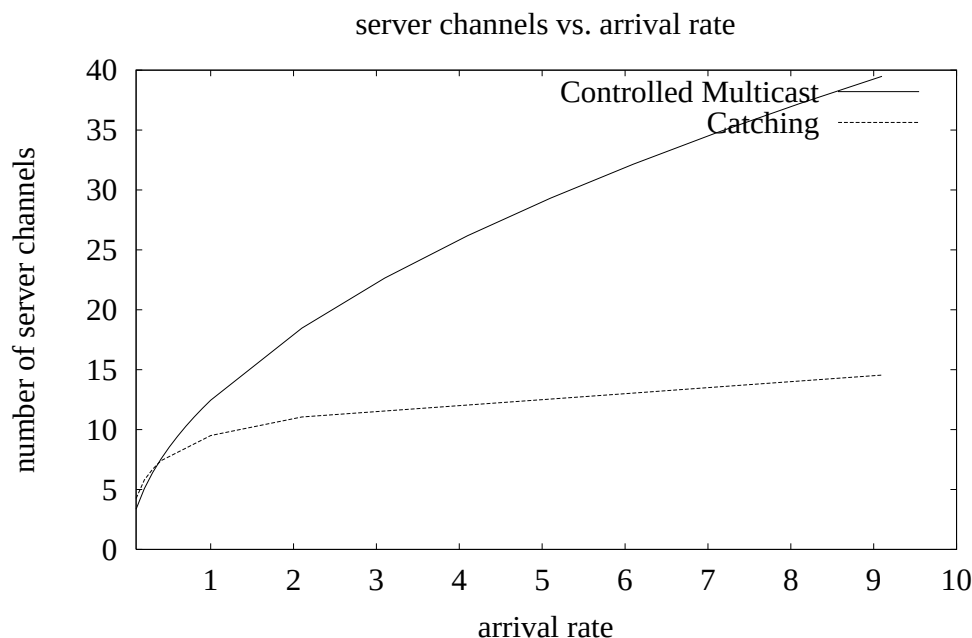


Figure 4: *Number of server channels vs. arrival rate.*

the expected number of channels required to deliver 90 minute long video object under catching and controlled multicast respectively, as the arrival rate of client requests for the video object varies from 0.1 to 0.9. We can see that catching requires fewer channels than controlled multicast when the request rate is greater than 0.4. When the request rate drops below 0.4, controlled multicast has better performance. This is because for a “cold” video, client requests for the video object do not come frequently enough to take advantage of periodic broadcast.

Although the analysis in this section is carried out based on the assumption that clients only have sufficient network bandwidth to support two channels, and that clients have sufficient disk storage space to store half of a video object, these assumptions are not essential. We can easily extend our analysis to cases where clients can support more than two channels by choosing the appropriate partition functions [7]. Furthermore, since the amount of disk storage space required at clients equals to the size of the largest broadcast segment [7], we can restrict the partition function in such a manner that the largest broadcast segment size is always smaller than the available client disk storage space (please refer to [7] to see how this can be done).

3.1 Optimality of Catching

In this section, we prove that the number of channels required by catching close to the minimum achievable by any broadcast scheme that supplies near-instantaneous service. Formally, for a video of length L and whose request arrival is a Poisson process with arrival rate λ , any broadcast scheme needs at least $\log(\lambda L)$ channels to supply the instantaneous service.

Consider the i -th frame of the video. First, we prove that we need at least $i + 1/\lambda$ channels for

broadcasting frame i .

Let random variable X_i denote the time between two consecutive i -th frame multicast. Here we use one frame time as our time unit. We claim that $E[X_i] \leq i + 1/\lambda$.

Suppose frame i is broadcast at time t . Any client that arrives between time $t - i$ and t can retrieve the same frame multicast at time t . However, the first client that arrives after time t could not. Let s denote the time that the first client arrives after time t . Frame i has to be broadcast once between time t and time $s + i$ to ensure the continuous playback of the first client arriving after time t . Therefore, $X_i \leq s + i - t$. Since the arrival process is Poisson with rate λ , we know that $E[s - t] = 1/\lambda$. Therefore, $E[X_i] \leq i + 1/\lambda$.

Since we need to use at least $1/E[X_i]$ channels for the i -th frame, we need at least $i + 1/\lambda$ channels for broadcasting frame i . Summing all frames, we need at least

$$\sum_{i=1}^L 1/E[X_i] \leq \sum_{i=1}^L (i + 1/\lambda) \approx \log(\lambda L)$$

channels for broadcasting the whole video. Comparing with (11), we see that the number of channels required by catching is within a constant factor from the minimum achievable by any broadcasting scheme. The constant factor depends on the optimal K_i^* in (8), which in turn depends on the broadcasting scheme used.

4 Selective Catching

In the previous section we developed the catching video delivery technique and showed that it is most effective for “hot” video objects. In this section, we design a new video delivery scheme, referred to as *selective catching*, which combines catching and controlled multicast to account for diverse user access patterns of video objects.

The key issue in the design of selective catching is to determine when to apply catching and when to apply controlled-multicast. To address this issue, we present a simple and straightforward criterion for classifying video objects based on their access patterns. A video object is considered “hot” if the expected number of channels required to deliver it using catching is less than that required using controlled multicast. Otherwise, the video object is considered “cold”. More specifically, a video object i is considered “hot” if and only if

$$K_i^* + L_i \lambda_i / (2 \sum_{j=1}^{K_i^*} f(j)) > \sqrt{2L_i \lambda_i + 1} - 1 \quad (12)$$

where K_i^* is defined in (8).

This definition of “hot” and “cold” video objects is based on the assumption that there are always sufficient channels available (i.e. at least K_i^*) that can be dedicated to broadcast a video object i . The definition can be extended to the case where this is not true by allocating dedicated broadcast channels to only “hottest” video objects among the “hot” video objects (say, based on user access patterns). For simplicity of exposition, we will not consider this case.

Simulation Parameters	Default Value
Number of Videos	100
Request Rate (requests/min)	50
Video Length (minutes)	90
Number of Channels	700
Disk Size (minutes of data)	30
Skew Factor	0.271

Table 1: *Parameters chosen for the simulation.*

In the rest of this section we conduct simulations to demonstrate that selective catching using this simple policy for classifying “hot” and “cold” video objects have superior performance over catching and controlled-multicast in terms of both channel usage efficiency and service latency.

In our simulations, we assume that client requests arrive at a video server according to a Poisson distribution with an average rate λ (i.e., the average interarrival time between consecutive requests is $1/\lambda$). For a given request, the probability distribution of video selection obeys a Zipf-like distribution [11]: for a collection of N video objects, the probability of selecting video object i , $i = 1, 2, \dots, N$, is $F_i = g_i / \sum_{j=1}^N g_j$, where $g_i = \frac{1}{i^{1-\theta}}$. Here θ denotes the skew factor in video access patterns. In our simulations, we use $\theta = 0.271$. This value of θ is known to closely match the video popularity distributions observed by video rental stores [2].

Unless noted otherwise, the workload and system parameters chosen for our simulations are listed in Table 1. Each run of our simulations simulates 150 hours of client requests. In our simulations, we dedicate a fixed number (K_i^*) of broadcast channels for each “hot” video objects. The remaining channels are used either for controlled multicast or for delivering “catch-up” prefix streams. These channels are scheduled on a on-demand basis, and requests for these channels are served in First-Come-First-Serve order.

Figure 5 compares the expected service latency experienced by clients under controlled multicast, catching and selective catching, as the total number of channels varies from 600 to 1,000. The average arrival rate, λ , of client requests is fixed at 50 requests per minute. From the figure, we see that catching performs worse than controlled multicast when the number of total channels is small. This is because catching dedicates a fixed number of broadcast channels for each video objects no matter whether it is “hot” or “cold”. Hence when the total number of channels is small, there are not sufficient channels left for delivering “catch-up” streams. As a result, clients may experience large service latency. But as the number of server channel increases, this problem becomes less severe. In these cases, catching has significantly better performance than controlled multicast. Selective catching, however, attains the best performance among the three in all cases. In particular, we observe that with 710 or more channels, selective catching achieves *zero* expected service latency. To obtain the same result, controlled multicast requires at least 900 channels.

Figure 6 shows the effect of the arrival rate on the expected service latency for controlled multicast, catching and selective catching. In this simulation, the request arrival rate varies from 40 to 80 requests per minute, and the total number of channels is fixed at 700. From the figure we see that in

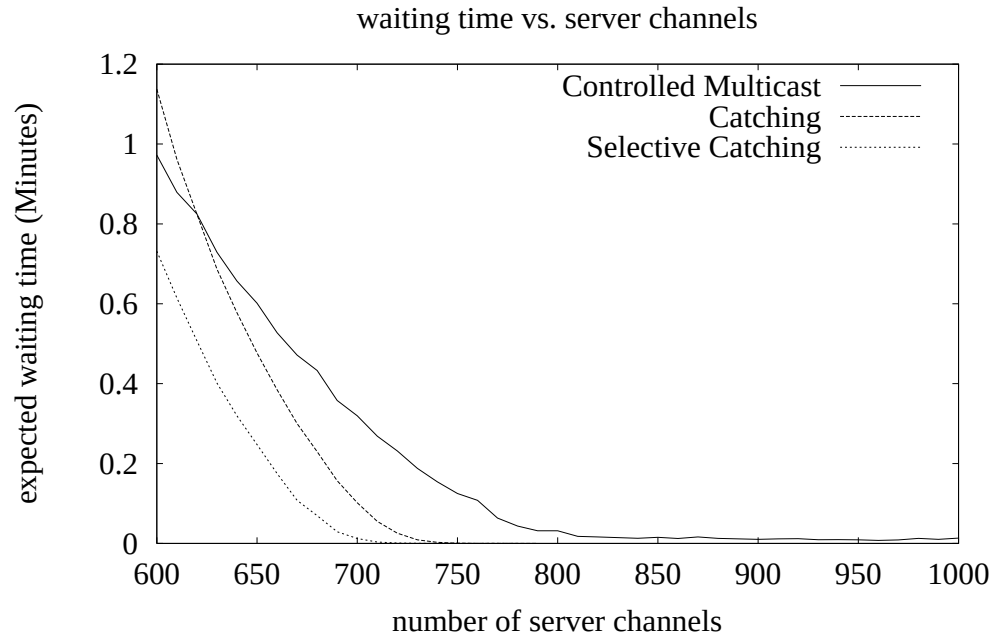


Figure 5: *Expected waiting time vs. number of server channels.*

most cases, catching and selective catching outperform controlled multicast. In particular, as the arrival rate increases from 40 to 80, the benefit of catching and selective catching becomes significant. As the arrival rate increases further, we expect that catching and selective catching has the same performance since all videos become hot. However, we do not expect the VOD system to perform in the region where the arrival rate is greater than 80 when the number of server channels is 700, since the service latency is unacceptably high and our optimization criteria is based on the assumption that the expected service latency is close to zero. Also note that selective catching can provide a zero expected service latency even when the request arrival rate is 50 requests per minute, while catching can provide a zero expected service latency only when the request arrival rate is 40 requests per minute.

Figure 7 compares the total expected number of channels required for controlled multicast, catching and selective catching as a function of the request arrival rate. In this simulation, we vary the request arrival rate from 40 to 100 requests per minute. The results show that selective catching requires at least 150 channels fewer than that required by controlled multicast, while catching requires at least 100 fewer channels. As the request arrival rate increases, more video objects become “hot”. As a result, the difference between the number of channels required by selecting catching/catching and controlled multicast is further widened. In general, selective catching requires few channels than catching, and the difference between them diminishes as the request arrival rate increases. As the request arrival rate reaches close to 100, the expected number of channels required by catching approaches to that required by selective catching. This is because as all videos become “hot”, selective catching coincides with catching.

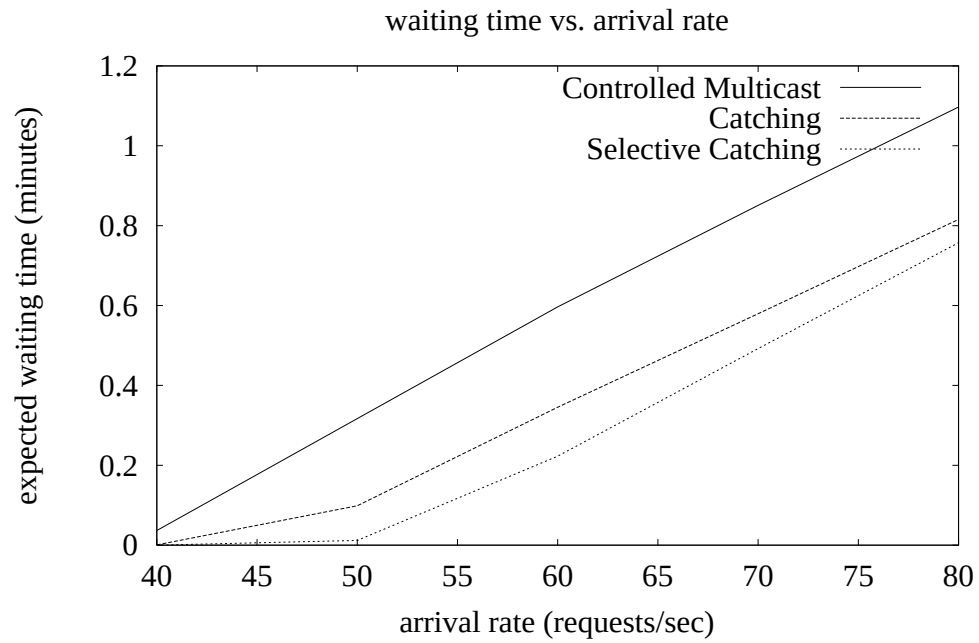


Figure 6: *Expected waiting time vs. arrival rate.*

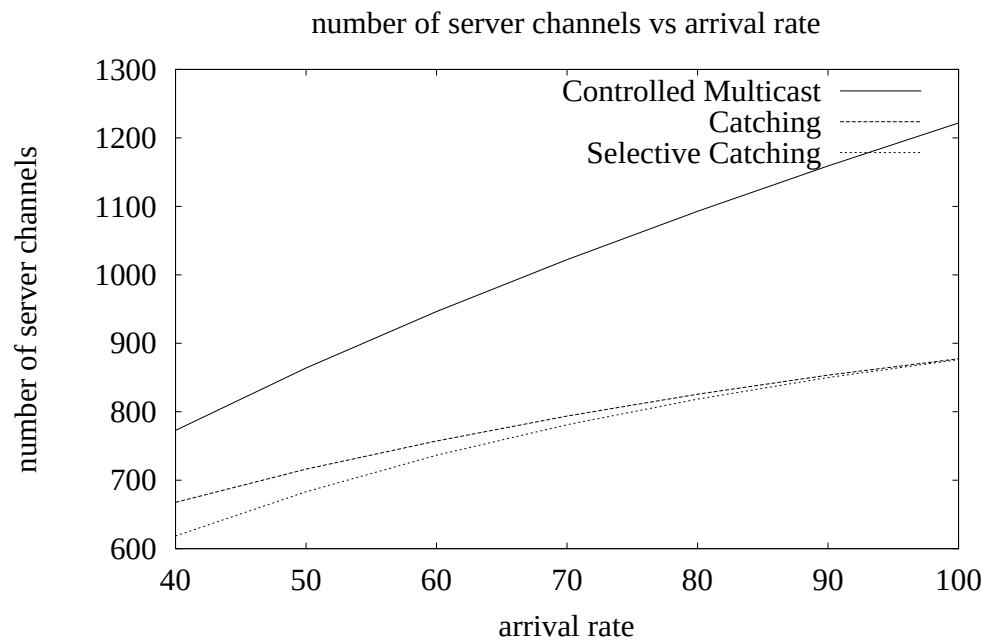


Figure 7: *Expected number of server channels vs. arrival rate.*

5 Proxy-Assisted Selective Catching

The selective catching video delivery technique is particularly amenable for deployment in a proxy-assisted video delivery environment [12, 13]. In such an environment, proxy servers are strategically placed between, say, local access networks and the backbone wide-area network (see Figure 8 for an illustration). This proxy-assisted video delivery environment has several advantages over the traditional stand-alone video server environment (see, e.g., [12]) for discussion on this regard). In this section we present a *proxy-assisted catching* video delivery scheme based on selective catching and demonstrate through an empirical study that the proposed proxy-assisted catching can drastically reduce the total number of channels required at the central video server. Furthermore, the expected service latency of clients can be significantly improved.

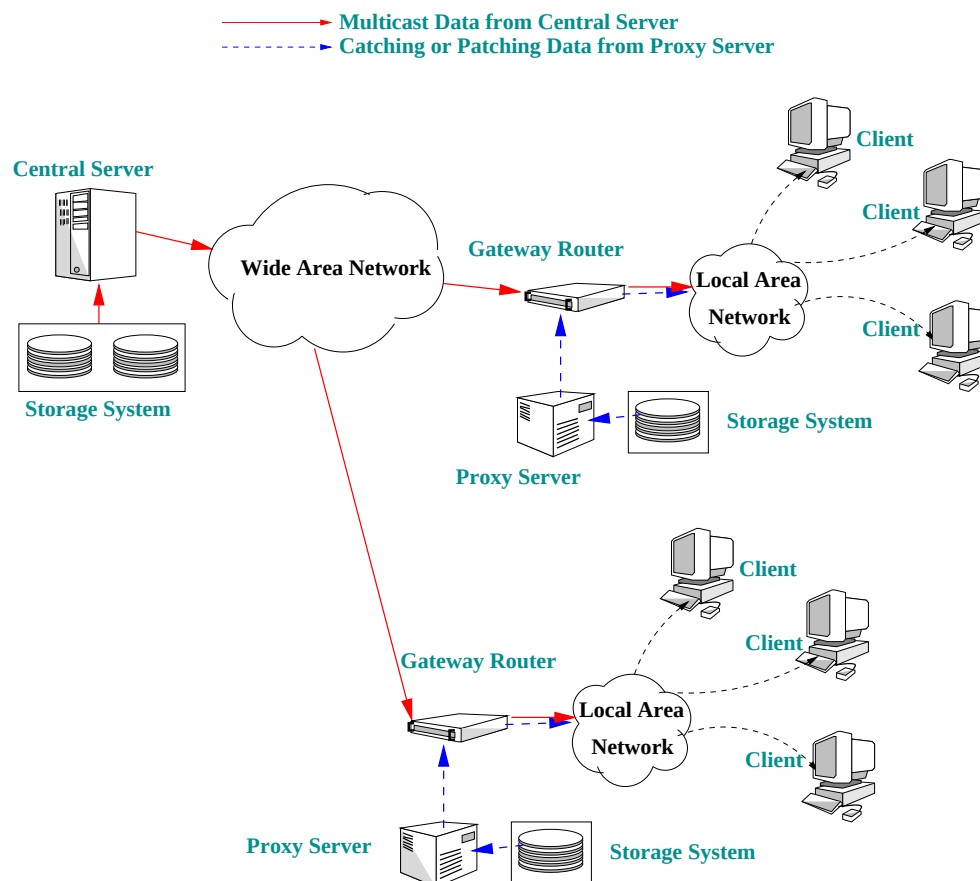


Figure 8: An overview of proxy-assisted catching architecture.

In our proxy-assisted catching scheme, we stage, i.e., pre-store the first broadcast segments (i.e., the prefixes of length F_i for video i) of some or all “hot” video objects as well as the initial segments (determined by the threshold T_i in controlled multicast) of some or all “cold” video objects. In this paper, we will ignore the issues such as how much proxy storage space is needed to stage videos and which video prefixes should be staged in proxy servers (these will be the subject of a future paper). We will focus instead on the design of our proxy-assisted catching scheme and on the study of the efficacy of our proxy-assisted catching scheme.

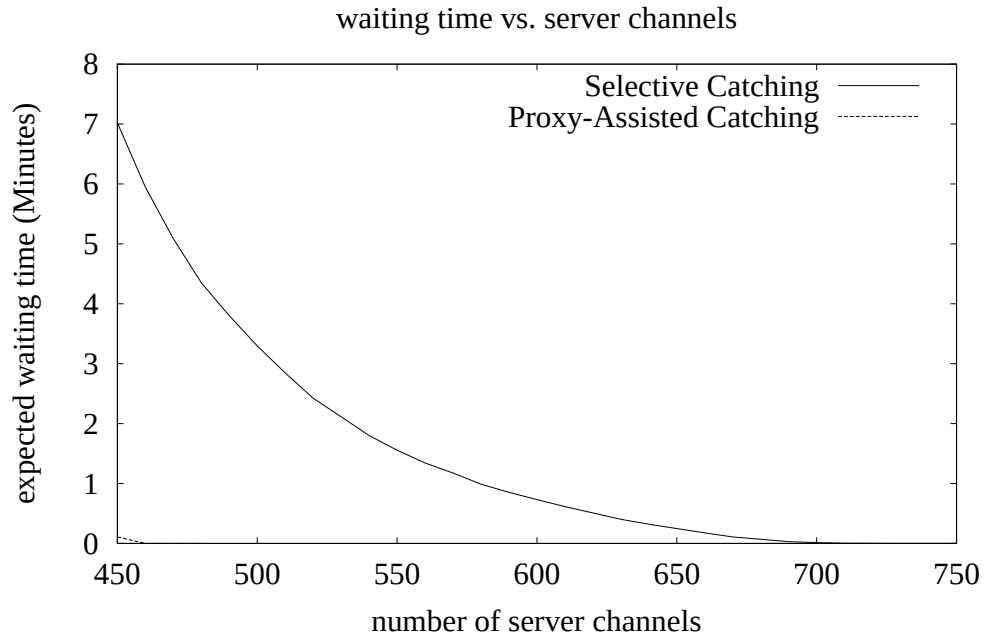


Figure 9: *Expected waiting time vs. number of server channels.*

We distribute the responsibility of video delivery between the central video server and proxy servers to leverage the processing power and locality of proxy servers. In other words, we attempt to significantly reduce and potentially eliminate service latency for clients while in the same time taking advantage of the resources available at proxy servers. In the following, we describe the central server, proxy server and client algorithms used in our proxy-assisted selective catching.

Central Server Algorithm:

In the case of catching, the central server is responsible for broadcasting “hot” video objects using a fixed number of dedicated broadcast channels. Similarly, in the case of controlled multicast, the central server is always responsible for delivering a complete video stream via a multicast channel.

For both “hot” and “cold” video objects, if the prefix of a video object requested is not stored in the proxy server to which the client is “designated”, the server is also responsible for delivering the prefix of the video object to the client.

Proxy Algorithm:

We assume that a client request for a video object is always sent to a proxy server designated for the client. When a client request arrives at a proxy server, it will contact the central server to obtain the necessary channel status information such as the latest broadcast period (in the case of catching), the latest multicast starting time (in the case of controlled multicast), and channel number, etc., and forward this information to the client. In addition, proxy servers are responsible for delivering “catch-up” streams (in the case of catching) or “patch-up” streams (in the case of controlled multicast) via local unicast channels, whenever the prefixes of the requested video objects are available at their local video prefix stores.

Client Algorithm:

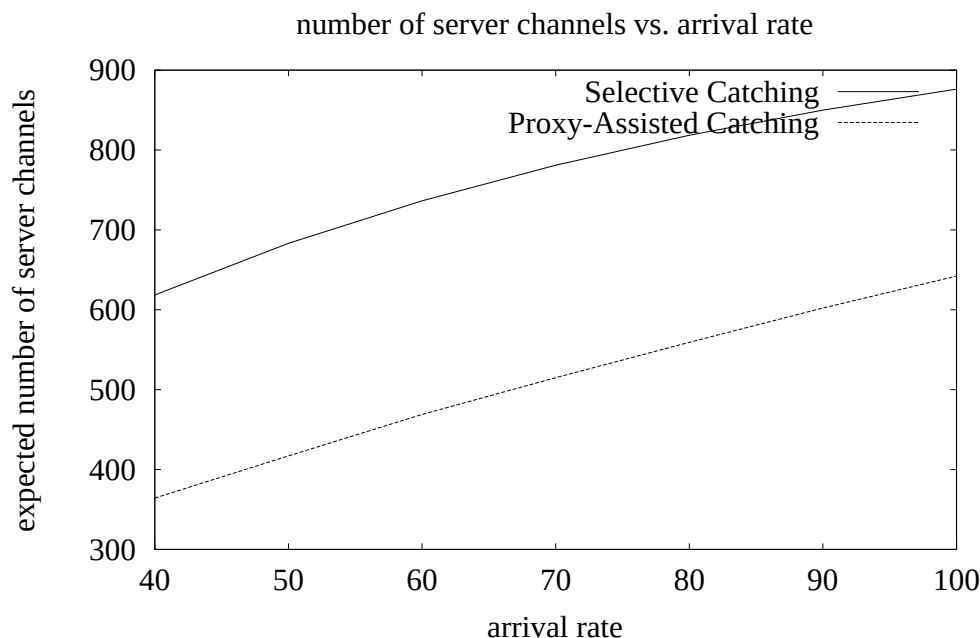


Figure 10: *Expected number of server channels vs. arrival rate.*

When a client wishes to request a video object, it first contacts its designated proxy server. From this proxy server, it receives the channel status information and joins in the appropriate channel(s) using a prescribed schedule. In addition, if the prefix of the requested video by the client is staged at its designated proxy server, the client will receive the missing portion of the prefix via a local unicast channel from the proxy server. Otherwise, the client is instructed to contact the central server to set up a unicast channel from the central server and retrieves the missing portion of the video prefix via this channel.

We now demonstrate the efficacy of our proxy-assisted selective catching scheme based on simulations. In our simulations, we make the same assumptions and use the same simulation parameters (wherever applicable) as in Section 4. In addition, we also assume that we have sufficient proxy resources to store prefixes for all videos. Specifically, we assume that each proxy server has 40GBytes of storage space and a disk I/O bandwidth of 88 Mbit/s⁴.

Figure 9 compares the expected service latency of selective catching and proxy-assisted selective catching, as the number of channels available at the central server increases. We see that that proxy-assisted catching can provide an expected service latency close to 0 with only 460 server channels. Whereas, to provide the same service, selective catching needs 700 channels. Proxy-assisted selective catching yields a 36% saving in the number of channels required at the central server. Figure 10 plots the expected number of channels required at the central server as a function of the request arrival rate. Proxy-assisted selective catching achieves significant reduction in the number of central server channel requirement in all the range of the request arrival rates.

In the next set of simulations, we study the resource tradeoff between the central server and the proxy server. We assume that the proxy server has limited amount of the storage space. Consequently, we only

⁴These assumptions are not unreasonable with current trend in disk storage technologies.

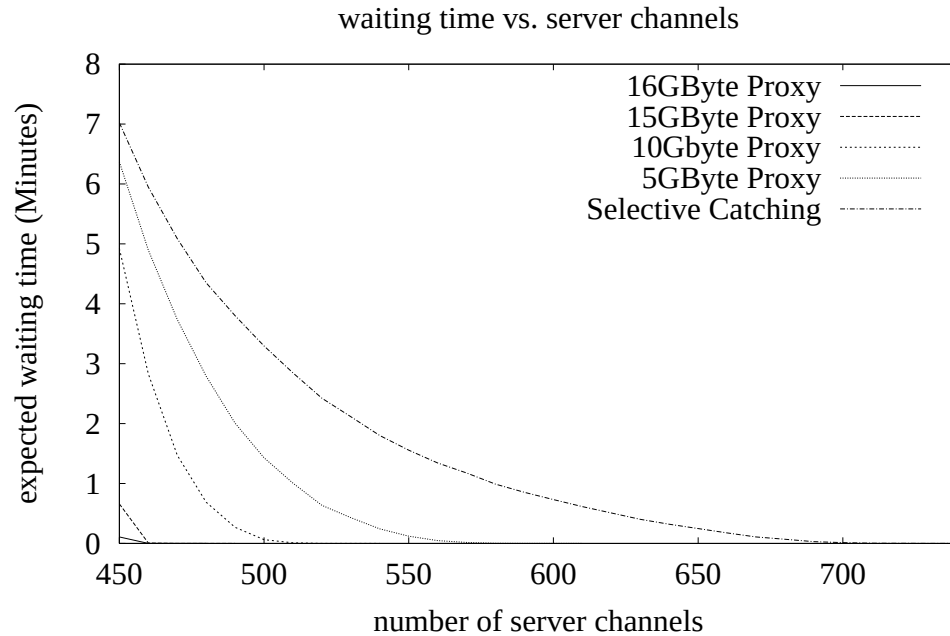


Figure 11: *Expected waiting time vs. number of server channels.*

stage the prefixes of the most popular (i.e. “hottest”) videos for a given storage constraint. In Figure 11 we show the performance of proxy-assisted selective catching as the amount of storage space at the proxy server varies among 5 GBytes, 10GBytes, 15GBytes and 16 GBytes. The expected service latency of proxy-assisted selective catching under these proxy storage constraints as well as that of selective catching are plotted. Comparing with the performance of selective catching, we see that with relative small amount (e.g., 5 GBytes) of proxy storage space, there still is a significant reduction in the expected service latency using proxy-assisted catching. Hence, the advantage of proxy-assisted selective catching does not critically hinge on the availability of proxy storage space.

6 Conclusions

In this paper we have presented a novel and efficient video delivery technique called catching. This technique takes advantage of the resource efficiency of the periodic broadcast based “server-push” schemes while providing near-instantaneous service to a large number of clients. This technique is particularly effective for “hot” (i.e., frequently accessed) video objects. In order to account for the diverse access patterns for a collection of video objects in a video server, we have also designed an efficient video delivery scheme called selective catching. This scheme combines the catching technique for delivery of “hot” video objects with the controlled multicast technique for delivery “cold” video objects. Through simulations we demonstrate that the proposed selective catching can achieve superior performance over existing techniques in terms of both server/network resource requirements and service latency. We have also extended the proposed selective catching scheme to a proxy-assisted video delivery environment. Simulations are carried out to illustrate the significant reduction in server and network resource require-

ments achieved using this proxy-assisted video delivery architecture.

References

- [1] C.C. Aggarwal and J.L. Wolf and P.S. Yu. "A Permutation-Based Pyramid Broadcasting Scheme for Video-on-Demand Systems". *Proc. of the IEEE Int'l Conf. on Multimedia Systems*. June 1996.
- [2] C.C. Aggarwal and J.L. Wolf and P.S. Yu. "On Optimal Batching Policies for Video-on-Demand Storage Server". *Proc. of the IEEE Int'l Conf. on Multimedia Systems*. June 1996.
- [3] S. Viswanathan and T. Imielinski. "Metropolitan Area Video-On-Demand Service Using Pyramid Broadcasting". *IEEE Multimedia Systems*. 4:197-208, 1996.
- [4] K.A. Hua and S. Sheu. "Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-Demand Systems". *ACM SIGCOMM*. Sept. 1997.
- [5] K.A. Hua and Y. Cai and S. Sheu. "Patching: A Multicast Technique for True Video-on-Demand Services". *ACM Multimedia*. Sept. 1998.
- [6] A. Dan and P. Shahabuddin and D. Sitaram, "Scheduling policies for an on-demand video server with batching". In *Proc. of ACM Multimedia*, October 1994, pp. 168-179.
- [7] L. Gao, J. Kurose, D. Towsley, "Efficient Schemes for Broadcasting Popular Videos", *Proceedings of NOSS-DAV*, Cambridge, UK, July 1998.
- [8] L. Gao, D. Towsley, "Supplying Instantaneous Video-on-Demand Services Using Controlled Multicast," *Proceedings of IEEE Multimedia Computing and Systems*, Florence, Italy, June 1999.
- [9] D.L. Eager and M. K. Vernon, "Dynamic Skyscraper Broadcasts for Video-on-Demand", *Lecture Note of Computer Science*, Sep. 1998.
- [10] D.L. Eager, M.C. Ferris, and M.K. Vernon, "Optimized Regional Caching for On-Demand Data Delivery", In *Proc. of MMCN*, Jan. 1999.
- [11] G. Zipf, *Human Behaviour and the Principle of Least Effort*, Addison-Wesley, 1949.
- [12] Y. Wang, Z.-L. Zhang, D. H.C. Du, and D. Su, "A Network-Conscious Approach to End-to-End Video Delivery Over Wide Area Networks Using Proxy Servers", In *Proc. of INFOCOM*, 1998.
- [13] S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams", In *Proc. of INFOCOM*, 1999.
- [14] W. Liao, V.O.K. Li. "The split and merge (SAM) protocol for interactive video-on-demand systems" *Proc. of INFOCOM'97*, p. 1351.

A Determining the Optimal Number of Dedicated Channels

In this appendix, we describe how to determine the optimal number of dedicated broadcast channels, i.e., K^* defined in (8). To proceed, we define $h(n) = \sum_{i=1}^n f(n)$. It is not hard to verify that

$$h(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2, & \text{if } n = 2 \\ 3, & \text{if } n = 3 \\ 5, & \text{if } n = 4 \\ 7, & \text{if } n = 5 \\ 5^{(n-6)/4} * 27/2 - 1.5, & \text{if } n > 5, n \bmod 4 = 2 \\ 5^{(n-7)/4} * 37/2 - 1.5, & \text{if } n > 5, n \bmod 4 = 3 \\ 5^{(n-8)/4} * 61/2 - 1.5, & \text{if } n > 5, n \bmod 4 = 0 \\ 5^{(n-9)/4} * 85/2 - 1.5, & \text{if } n > 5, n \bmod 4 = 1 \end{cases}$$

Note that for $j = 0, 1, 2, 3$, $1/h(4k + j + 2)$ is convex in k , $k \geq 0$. Therefore, there exists a $k \geq 0$ such that $2 + (4k + j) + L_i/h(2 + (4k + j))$ is minimized. Let k_j^* denote this value of k . Then $K_i^* \in 1, 2, 3, 4, 5, 4k_0^* + 2, 4k_1^* + 3, 4k_2^* + 4, 4k_3^* + 5$. Determining K_i^* is thus quite straightforward since k_j^* , $j = 0, 1, 2, 3$, can be easily derived from the convexity of $1/h(4k + j + 2)$.