Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-029

An Efficient Client Collaboration Framework for Large-Scale Streaming

Ewa Kusmierek, Yingfei Dong, and David Du

August 08, 2005

Loopback: Exploiting Collaborative Caches for Large-Scale Streaming

Ewa Kusmierek
Poznan Supercomputing and Networking Center
91-704 Poznan
kusmiere@man.poznan.pl

Yingfei Dong
Dept. of Electrical Engineering
Univ. of Hawaii
Honolulu, HI 96822
dong@hawaii.edu

David Du*
Dept. of Computer Science
Univ. of Minnesota
Minneapolis, MN 55455
du@cs.umn.edu

Abstract

In this paper, we propose a *loopback* approach in a two-level streaming architecture to exploit collaborative client/proxy buffers for improving the quality and efficiency of large-scale streaming applications. At the upper level we use a Content Delivery Network (CDN) to deliver video from a central server to proxy servers. At the lower level a proxy server delivers video with the help of collaborative client caches. In particular, a proxy server and its clients in a local domain cache different portions of a video and form delivery loops. In each loop, a single video stream originates at the proxy, passes through a number of clients, and finally is passed back to the proxy. As a result, with limited bandwidth and storage space contributed by collaborative clients, we are able to significantly reduce the required network bandwidth, I/O bandwidth, and cache space of a proxy. Furthermore, we develop a local repair scheme to address the client failure issue for enhancing service quality and eliminating most required repairing load at the central server. For popular videos, our local repair scheme is able to handle most of single-client failures without service disruption and retransmissions from the central server. Our analysis and simulations have shown the effectiveness of the proposed Loopback Scheme.

EDICS: 5-STRM

I. Introduction

Video-on-demand has become one of the most popular applications over the Internet and many approaches to video streaming have been proposed in the past two decades. Generally, there are three basic types of video streaming architectures: an architecture based on the client-server model, Content Distribution Network (CDN) and most recently introduced Peer-to-Peer (P2P) approach. One of the main requirements with respect to a streaming architecture is scalability. A number of mechanisms have been proposed to improve the scalability of traditional client-server based schemes. These include batching [2], patching [10] and periodic broadcast [9]. Unfortunately, due to lack of multicast capability in current IP networks, these schemes are difficult to implement and client-server based solutions are so far not suitable for a large-scale operation. A CDN architecture on the other hand, is based on a number of CDN servers placed at the edges of the Internet. The video content is first distributed to these servers and then delivered by each CDN server to the clients in its neighborhood. Such an approach requires expensive

*Phone: (612) 625-2560, Fax: (612) 625-0572

investment for a broad delivery infrastructure and demands a rather large amount of resources to achieve scalability. Moreover, a CDN has fixed resources which makes it difficult to deal with the flash crowd problem [12], [19].

A P2P approach offers a possible solution but also presents some challenges. In a P2P scheme a client not only receives service but also offers service to other clients as a peer. The clients collaborate with one another to reduce the video server load and to improve the scalability. The capacity of a P2P system built by collaborating peers adjusts to the demand. The capacity increases with an increase in demand which allows us to address the flash crowd issue. However, the amount of resources contributed by a single peer, such as storage space and network bandwidth, is limited. The peers are usually willing to participate in video streaming only for a limited amount of time. Moreover, a P2P system is not as reliable as client-server and CDN based system, and does not offer quality assurance.

In this paper we propose and analyze a client collaboration mechanism for Video-on-Demand (VoD) streaming that allows us to efficiently utilize the P2P capacity. We assume that each client dynamically caches a portion of the video it receives and makes it available to other clients. The storage space contributed by a client is limited and may be sufficient to store only a small part of a video. A client is capable of delivering only one stream at a time and is willing to cooperate only during the video playback and for a short period of time after the playback ends. Thus, the resources offered by a client are limited and it is important to utilize them efficiently. Moreover, we assume that there is a certain probability that a client may fail or choose to leave while delivering a video to its peers.

Based on the above assumptions we propose a *Loopback* mechanism to address the challenges posed by a P2P approach. First, we organize client collaboration in a local environment with the help of a CDN server which we refer to as a proxy server. Thus, we assume that there exists a CDN to deliver video from a central server to a number of proxies serving local communities. A proxy server streams videos to the clients in its community with the assistance of a client-collaborative network formed with the *Loopback* scheme. Hence, we utilize a hybrid CDN-P2P architecture as proposed in [5] but our focus in this paper is on the *local client cooperation*. Building an overlay for client communication on a large scale is not very efficient due to a need for considering properties of the physical connectivity. In a local environment, on the other hand, such problems do not occur. It is the network bandwidth between the central server and a proxy server that may be limited and expensive. Therefore, it is important to utilize CDN bandwidth efficiently.

A proxy server often has limited resources and is unable to cache an entire video. Instead, a prefix of a video is cached [13], [14]. We show how client collaboration is organized to complement such type of

video caching by the proxy. In *Loopback*, clients arriving close to each other in time form a forwarding ring with the first client receiving data from the proxy and the last client returning data to the proxy. Thus, we call this group of clients a *loop*. A video stream is forwarded from one client to another within a loop in the order of their arrival. Each client dynamically caches only a small portion of a video. If a client buffer fills up before the next client arrives, the video frames are evicted from the buffer by either forwarding them back to the proxy or by simply discarding them if the proxy buffer space is also filled up. The next client unable to join the existing loop starts a new loop receiving data directly from the proxy or from the central server. We show that *Loopback* significantly reduces storage space and bandwidth requirements of the proxy and the load of the central server even if the resources provided by each peer are limited. Our scheme is especially efficient for videos with high popularity. For such videos, the proxy server needs to cache only a small portion of the video or none and uses only a few streams to service a larger number of clients. We address also the reliability of the *Loopback* with redundant caching. Our approach does not require any additional resources beyond the amount declared by clients for a regular operation.

The remainder of this paper is organized as follows. In Section II we summarize the related work on P2P-related approaches to video streaming. In Section III we describe the principles of *Loopback* operation. In Sections IV and V, we present a formal performance analysis of the *Loopback* approach for serving a single video. In Section VI we extend the analysis to include the performance of *Loopback* with multiple videos. In Section VII we analyze the influence of client failures and in Section VIII, we develop a local-repair mechanism to deal with the failures of collaborative caches for ensuring service quality. In Section IX we conclude this paper and discuss our future research.

II. Related Work

The idea of client collaboration has been widely applied to file sharing [16], [20], [17] and application level multicast for both live and on-demand video streaming services. Collaboration schemes utilize resources contributed by clients acting as peers and can be very efficient in reducing central server and network load. A number of issues have to be addressed to efficiently organized client collaboration. Collaborating clients may have limited resources such as upstream bandwidth and storage space and may be willing to offer services to other peers only for a limited amount of time. Their service may be also unreliable due to unannounced departures. In most cases a server keeps track of participating peers and is responsible for efficient resource utilization.

A number of solutions have been proposed to peer collaboration under different assumptions of peer capabilities. In P2Cast [7] a client may be capable of providing multiple streams. An application-level

forwarding tree consisting of a set of active clients in a session is constructed for VoD service. A session is formed by a group of clients arriving close in time. The first client in a session receives video directly from the server. Each next client receives the video from the server or from one of the preceding clients depending on bandwidth availability. In addition, the client receives also a patch stream for a part of the video already transmitted within the session. Each client caches an initial portion of a video and the patch source selection is the same as the selection for the base stream. In our scheme a client buffer content is not fixed. Each client caches certain amount of the most recently played data and makes this data available to the subsequent client. The amount cached depends on the client inter-arrival time. In this way there is no need for patch streams within a single session.

In P2cast one client may serve multiple peers simultaneously but its availability is required only for the duration of a video playback. In the hybrid architecture presented in [22] a client is capable of transmitting data at a fraction of the playback rate. Hence, a many-to-one delivery model is assumed. One client receives video stream from multiple peers. Client availability, on the other hand, is increased to a multiple of the video playback length so that one client may provide service to multiple peers. A client has to cache an entire video resulting in a substantial storage space requirements. In contrast, we assume the client storage space is limited to a small portion of a video and the client has to participate in streaming only for a short time after the end of video playback, namely the time required to stream the cached data. However, the client's upstream bandwidth (used to deliver frames to another client or to proxy) is assumed to be sufficient for only one video stream. The extended client availability considered in [22] increases the influence of a client failure on the performance. We do not assume that the peers provide a reliable service and propose a mechanism to prevent playback disruption.

Unreliability of peers is one of the most important challenges in organizing client collaboration. An early departure of one peer may affect potentially a large number of dependent peers resulting in playback disruptions and a need to reorganize the service. Playback disruption caused by an early departure of a client in P2Cast is prevented with playback delay that allows client buffer to build up, and with shifted forwarding. In [6] live video playback disruption is prevented with time shifted transmissions for video patching and initial buffering. A server provides the original video stream and a number of time shifted streams. A client experiencing service disconnection patches a portion of the video missed during time needed to reconnect to the multicast tree from an appropriate time shifted transmission. In CoopNet [15] which is designed for live streaming, client failure issue is addressed with redundancy of data and distribution paths. The data redundancy is achieved with Multiple Description Coding. Each client receives multiple description streams from a number of peers. The video content can be reconstructed

from a subset of all these streams. Thus, a failure of one peer, and consequently a discontinuity of one stream, does not disrupt the playback but may only affect its quality. Path redundancy is achieved with multiple application-level multicast trees, one for each video description. In *Loopback* we utilize buffer space contributed by each peer for redundant caching. Such an approach allows us to prevent playback disruption with probability that increases with the popularity of the content. In addition, the buffer is built up without playback delay.

Live streaming systems are often designed as an application-level multicast tree. The tree height affects the distribution delay which is an important parameter in live streaming. Thus, an efficient method for building an application level multicast tree is required. Such methods are presented in [4], [11], [3], [21], [15]. A peer selection process is faced for both live and VoD service. Given a number of candidates, a set of supplying peers for a new peer is selected based on bandwidth availability and some other network path characteristics such as delay used in P2Cast [7]. Promise with ColletCast [8] exploit the properties of the underlying network, topology and performance, for selection of supplying peers. A network tomography method is used to build a graph representing connectivity between candidate peers and the receiver. Each network segment is characterized by loss rate, available bandwidth and delay. A set of supplying peers and a set of standby peers are selected based on the collected information. The session is continuously monitored and a switch to another peer from a standby set is executed in case of a peer failure. We use a hybrid architecture [5] in which client collaboration takes place within a local community. In such an environment peer selection can be based on peer availability only since the network path characteristics is similar for all peers.

Our *Loopback* mechanism further exploits the idea of passing data along in clients as proposed in *chaining* [18] which is one of the early proposed P2P techniques. However, in *chaining*, a client may have to cache an entire video in the worst case, which is a rather high requirement for a common client. In this paper, we assume that each client contributes only a limited small buffer space for caching. In addition, the extended *chaining* scheme [18] reduces server resource usage by delaying client to extend the chain length. Thus, the improvement is achieved at the cost of an additional delay. In our scheme each client starts receiving data immediately. The fundamental difference between *chaining* and *loopback* is that both the proxy server and *client collaboration* play the key roles in our *Loopback* scheme. A client request always goes to the proxy server so that the proxy server can manage collaborative clients for caching data efficiently.

III. Client Collaboration with Loopback

In this section we describe how client collaboration is organized with the *Loopback* approach. First, we list the assumptions that we make about a collaborating client. We assume that the client caches only data it receives for playback. The buffer used for caching is sufficient to store a small portion of a video. Data cached in client buffer is available during the video playback and for a period of time after the playback. The upstream bandwidth (used for delivering frames to another client or loopback to the proxy) for a client is sufficient for only one transmission at the playback rate. Thus, each client contributes a small storage space and bandwidth. Initially we assume also that each client is reliable. The issue of client failure probability and quality assurance is addressed in Section VIII.

The proxy server in a community of clients is part of a CDN. The proxy caches video data obtained from the central server, delivers video to the clients in its local community and organizes clients into a collaborative network. The main goal of this collaboration is to reduce the amount of data required from the central server across WANs. The proxy storage space and bandwidth are limited. Therefore, the proxy is not capable of caching all data requested by its clients. However, at first we assume that the proxy caches an entire video and show how client collaboration can reduce the storage space and bandwidth requirements of the proxy. In this way we demonstrate how many proxy resources are required for a video of a given popularity so that no data has to be requested from the central server. Next, we analyze the case when the proxy caches the video only partially and some data potentially has to be requested from he central server. For this case we analyze the *Loopback* influence on the central server load.

All client requests for a video are directed to a proxy so that the proxy can manage clients for efficient data caching. The first client requesting a video receives data from the proxy. The video frames received are played and cached in the client's buffer for a limited time with a limited buffer space. Once a frame is transmitted, the proxy discards the frame to make room for other streams. If the next client requesting the same video arrives before the first client's buffer fills up, the frames are streamed to the newcomer. The frames that have been already transmitted are removed from the buffer to make room for the subsequent frames. If the next request for the video does not arrive on time and the buffer of the first client becomes full, the oldest frames are passed back to the proxy and evicted from the buffer in this way. Recall that for now we assume that proxy has enough storage space to cache the entire video.

If a video is popular, there are a number of clients whose requests arrive before their respective previous client buffer fills up. Then, each of these clients receives the video from the previous client in the order of their arrivals, and forwards the video to the next client. The clients form a forwarding ring with the first client receiving data from the proxy. Eventually the next request for the video arrives after the previous

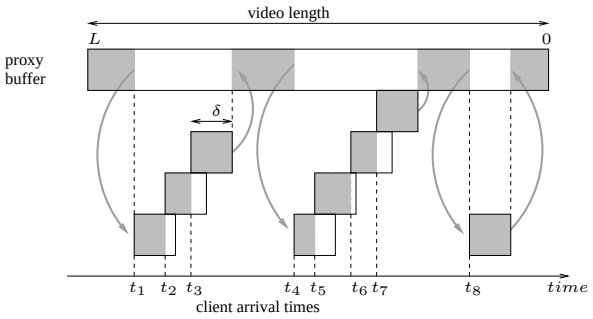


Fig. 1. Illustration of Client *Loops*.

TABLE I

NOTATIONS

L	video length
δ	fixed buffer size contributed by a client
λ	mean client arrival rate
f_c	expected client failure probability
t_i^j	arrival time of client i in loop j
$t_{i,b}$	start time of buffering for client i
$t_{i,e}$	end time of buffering for client i
b_i^j	buffer occupancy at client i
n^j	number of clients in loop j
d_s	delay in receiving data from a server

client buffer fills up. Then, the data is passed back to the proxy closing the forwarding *loop*. The late newcomer starts a new loop and receives the video from the proxy. Given a substantial length of the video, there may exist a number of loops at the same time. The proxy server is a member of each loop and each loop contains at least one client beside the proxy.

Fig. 1 illustrates three loops formed by eight client requests for a single video, consisting of three, four and one client, respectively. Each square represents a small client buffer. The shaded areas show the distribution of different portions of video data at the clients and at the proxy. The arrows show the directions of data flows between the proxy and each loop. Clearly, no data sharing and no conflicts exist between loops, i.e., a loop always returns data to the proxy before the first client in the next loop needs the data.

We assume that the proxy does not maintain a copy of a frame after transmitting to a client. The frames are passed from one client to another in a loop and eventually returned to the proxy. As a result, the entire video is dynamically distributed among the clients and the proxy. The amount of data cached by clients depends on the popularity of a video, i.e., the arrival rate of requests for the video. If the demand is high, there are few long loops containing many clients. Potentially all clients join the same loop for a very popular video. In this case the entire video may be cached by the clients releasing the proxy of storing any part of the video. In addition, the proxy does not use any bandwidth for this video except for bandwidth used to deliver the video to the first client. If the demand for a video is medium, the number of loops is larger and the number of clients in a loop is smaller. The proxy dynamically caches chunks of frames distributed throughout the entire video alternated with the chunks of frames cached by collaborating clients. The proxy only uses the bandwidth needed to forward one stream to each loop and to receive one stream from each loop. Hence, the bandwidth requirement is significantly reduced. Overall, for a popular video, proxy can use a few streams to service a large number of clients

with a low I/O bandwidth and only a small portion of a video or none cached by the proxy.

IV. Loopback Analytical Model

In this section, we analyze the basic properties of *Loopback* scheme. We describe the aggregate storage space contributed by collaborating clients and show its dependency on video popularity. We assume that a proxy server is capable of storing as much of a video as necessary to avoid data transfer from the central server. Our goal is to analyze the resource usage at the proxy for a single video under a given client arrival process. We derive the expected proxy buffer utilization and the expected proxy I/O bandwidth. In this way we obtain the resource requirements that would eliminate the high cost of transferring data through WAN. In this analysis, we also assume that there are no client failures.

Let us first introduce the notations used for the performance analysis of *Loopback* scheme. Let δ be the buffer size at each client and t_i denote the arrival time of the i^{th} client. In order to simplify the notations without loss of generality, we assume that the transmission rate and the playback rate of a video are equal to 1. Then if $t_i - t_{i-1} \leq \delta$, the i^{th} client receives the video from $(i-1)^{th}$ client; otherwise, it receives the video from the proxy. In the latter case, the $(i-1)^{th}$ client starts forwarding video back to the proxy at time $t_{i-1} + \delta$. Note that each client in the loop except for the last one may buffer less than δ. Let n^j be the number of clients participating in the j^{th} loop and t_i^j be the arrival time of the i^{th} client in the j^{th} loop. In some cases, we may drop the superscript j indicating loop index to simplify the notation when analyzing the performance of clients within a single loop. We assume that the client inter-arrival time X is exponentially distributed with the mean equal to $\frac{1}{\lambda}$. The notations are summarized in Table I.

A. Proxy Buffer Space vs. Aggregate Loop Buffer Space

The amount of data cached by a proxy for a given video depends on the number of clients receiving the video simultaneously. The proxy bandwidth usage depends on the number of concurrent loops. Thus, in the following, we first analyze the average loop length and the expected number of loops based on the mean client arrival rate. As a result, we are able to estimate how much data is distributed among clients in the concurrent loops of a video.

We consider two types of loops: single-client loops and multi-client loops. The distinction is made due to the fact that the amount of data cached by clients in a multi-client loop is smaller than δ with the exception of the last client. The last client in each loop, including one-client loops, caches exactly δ amount of data. The arrival rate of one-client loops is equal to $\lambda e^{-2\lambda\delta}$. Therefore, the average number of concurrent one-client loops is $(L+\delta)\lambda e^{-2\lambda\delta}$, where L is the video length. Recall that the last client has

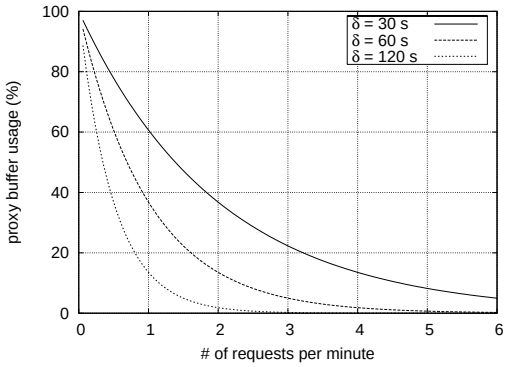


Fig. 2. Proxy disk usage

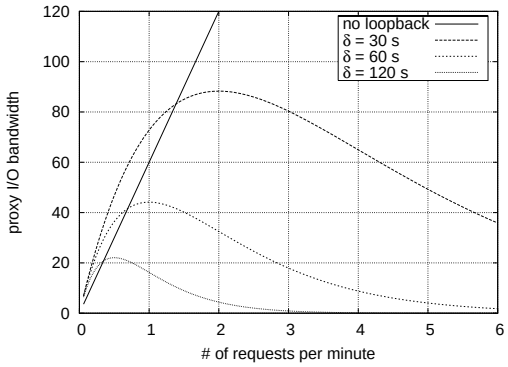


Fig. 3. Proxy I/O bandwidth usage

to return its buffer content to the proxy at δ time after playback. The average amount of data buffered by one-client loop is $\frac{\delta L}{L+\delta}$. Thus, the average amount of data distributed among one-client loops is equal to $L\lambda\delta e^{-2\lambda\delta}$. An n-client loop caches an amount of data equal to $\delta + (t_n - t_1)$ and the average amount of data cached by a multi-client loop is equal to $\frac{L(\delta+\hat{t})}{L+\hat{t}+\delta}$, where $\hat{t}$ is the mean difference between the arrival time of the last and the first client given by:

$$\hat{t} = \frac{1 - \lambda\delta e^{-\lambda\delta} - e^{-\lambda\delta}}{\lambda e^{\lambda\delta}(1 - e^{-\lambda\delta})} \tag{1}$$

The arrival rate of multi-client loops is derived as $\lambda(1-e^{-\lambda\delta})e^{-\lambda\delta}$. Then, the average number of concurrent multi-client loops is equal to $(L + \hat{t} + \delta)\lambda(1 - e^{-\lambda\delta})e^{-\lambda\delta}$ and the average amount of data cached by these loops is $L(\hat{t}+\delta)\lambda(1-e^{-\lambda\delta})e^{-\lambda\delta}$. Based on the above analysis, the average amount of data cached by the clients at a given time is estimated as:

$$B = L\lambda e^{-2\lambda\delta}\delta + L(\hat{t} + \delta)\lambda e^{-\lambda\delta}(1 - e^{-\lambda\delta}) \tag{2}$$

Fig.2 presents the amount of data cached by the proxy as a function of client arrival rate for three different values of client buffer size δ. The length of a video is set to 3600 s and the buffer sizes are 30 s, 60 s and 120 s. We observe that the amount of data cached by the proxy decreases as the client arrival rate increases. Each client caches certain amount of data, thus, the larger the number of clients receiving the video, the less data the proxy buffers. For buffer size equal to 120 s and the arrival rate of one client per four minutes, the proxy needs to cache around 55% of the video. When the arrival rate increases to one client per two minutes the proxy needs to cache less than 40% of the video. The proxy buffer occupancy eventually approaches 0 when all clients join the same loop and the entire video flows through this loop.

Increasing client buffer size also decreases the amount of data cached by the proxy. With the arrival rate of one client per two minutes, the proxy needs to cache about 60% and 40% of the video when the

clients cache 60 s and 120s of video respectively. We observe that for a high demand, the amount of data cached by the proxy for the given video is significantly reduced.

B. Proxy I/O Bandwidth

The proxy I/O bandwidth usage is given by a number of streams originating at the proxy and forwarded back to the proxy. The average number of streams due to a one-client loop is equal to $\frac{2L}{L+\delta}$ and the average number of streams due to a multi-client loop is equal to $\frac{2L}{L+\hat{t}\delta}$, where $\hat{t}$ is given by Eq. 1. The average I/O bandwidth usage is proportional also to the number of concurrent one-client and multi-client loops: $(L+\delta)\lambda e^{-2\lambda\delta}$ and $(L+\hat{t}+\delta)\lambda(1-e^{-\lambda\delta})e^{-\lambda\delta}$, respectively. Note that the total number of loops is equal to $(L\lambda-1)e^{-\lambda\delta}+1$ and approaches 1 as the arrival rate increases. The average I/O bandwidth due to all concurrent loops is equal to :

$$\hat{P} = 2L\lambda e^{-\lambda\delta} \tag{3}$$

Fig. 3 shows the proxy I/O bandwidth usage as a function of client arrival rates for three different sizes of client buffer. The results presented in Fig. 3 correspond to the results presented in Fig.2. Video playback rate is used as a unit for I/O bandwidth usage. For a given buffer size, the I/O bandwidth usage initially increases almost linearly due to an increase in the number of mostly one-client loops. As the arrival rate keeps increasing, the number of clients in a loop increases but also the number of loops increases resulting in further increase in I/O bandwidth usage. The maximum usage equal to $2L\frac{1}{\delta e}$ is reached for $\lambda = \frac{1}{\delta}$. From this point on the average loop length increases and the number of loops decreases. Thus, we observe a decrease in the I/O bandwidth usage. The usage eventually approaches 0. We observe also that the larger the client buffer is, the lower the I/O bandwidth usage at the proxy.

Fig. 3 shows also the proxy I/O bandwidth usage without *Loopback* mechanism. It equals $L\lambda$ due to a transmission of one video stream per client. Although initially the I/O bandwidth usage without *Loopback* is lower than that with Loopback, the situation changes very quickly with an increase in the client arrival rate. Clearly, *Loopback* significantly reduces the requirement of proxy I/O bandwidth for popular videos.

V. Loopback Model with Limited Proxy Buffer

We now relax the assumption that the proxy server can store an entire video. Since proxy has fewer resources than the central server, thus, it is impossible to cache all videos of interest. For some videos the proxy may be able to cache only a portion of the video. We first describe *Loopback* operation with partial caching provided by the proxy. Next, we examine how much data is available locally, i.e., the size

of the aggregate cache space of the proxy and the collaborating clients for a given client arrival process and analyze the central server load due to a given video. We show that *Loopback* significantly reduces the server load and further describe the ability of the system to adapt to variable video popularity.

A. Loopback Operation with Limited Proxy Buffer

We first describe how client collaboration is organized provided that the proxy caches only a portion of a video. We assume that the storage space of the proxy dedicated to a given video is equal to αL, where $0 \leq \alpha \leq 1$ and L is the video length. In common proxy caching schemes, the prefix of a video is cached by proxy server. Each client receives the prefix from the proxy and the remaining portion of a video from the central server. Similarly, in the *Loopback* scheme we assume that the proxy initially caches a prefix of the video.

When the request arrival rate is very low, there is usually only one client receiving the prefix from the proxy. The prefix frames are returned to the proxy after the playback and the client requests the suffix of the video from the central server. Hence, the operation without and with the *Loopback* mechanism is very similar in this range of client arrival rate. As the client arrival rate increases, multiple loops receive and return data to the proxy at the same time. Since each loop caches some data received from the proxy, it decreases the amount that needs to be cached by the proxy. There is room (i.e., proxy has reserved αL buffer space for this video) in the proxy buffer for the loops to return more than just the initial prefix data. Fig.4 shows an example of such a behavior. The first client in the first loop has already played the prefix and is receiving the subsequent frames from the server. Since the second loop has removed some frames from the proxy buffer and made more buffer space available, the first loop can return data *beyond the prefix* to the proxy. The amount of this additional data is equal to the amount of data buffered by the second loop. Note that the second loop will not only receive the prefix from the proxy but will also receive the additional amount of data that is deposited in the proxy buffer by the first loop.

In a general case, the earliest loop receiving data from the proxy can return the amount of data equal to the aggregate amount buffered by all subsequent loops. As a result, this additional amount of data is available from the proxy to the subsequent loops. The video prefix currently available locally becomes larger than the initial prefix of size αL. The larger prefix is buffered partially by the proxy and partially by the clients.

As long as there is data buffered by neither the proxy nor the clients, some frames have to be requested from the central server. Recall that the amount of data buffered by the clients increases with the arrival rate. For some value of the arrival rate, the clients buffer at least $(1 - \alpha)L$ and the community becomes independent of the central server since the entire video is available locally. The proxy buffers chunks of

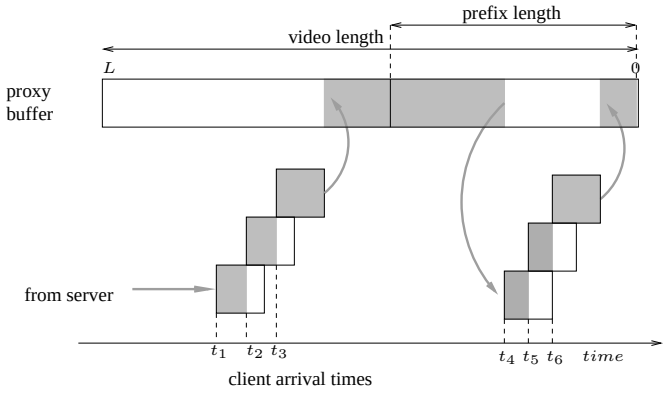


Fig. 4. Client "loops" with limited proxy space.

frames distributed throughout the entire video and the cumulative size of these chunks is smaller than or equal to the proxy buffer space reserved for this video: αL.

B. Local Data Availability

We now evaluate how much data is available locally for a given client arrival rate under the limited proxy buffer assumption. Let $\hat{w}$ denote the average amount of data buffered by a loop:

$$\hat{w} = (1 - e^{-\lambda\delta}) \min(L, \hat{t} + \delta) + e^{-\lambda\delta}\delta \tag{4}$$

where $\hat{t}$ is the average difference between the arrival time of the last and the first client in a multi-client loop given by Eq. 1. The average loop inter-arrival time is:

$$\Delta t = \frac{1}{\lambda e^{-\lambda\delta}} \tag{5}$$

Let k be the number of loops such that the total amount of data in the proxy buffer (separating the loops) is no larger than the reserved buffer space: $k(\Delta t - \hat{w}) \leq \alpha L$. The following condition also has to be satisfied: $k\Delta t \leq L$. Thus,

$$k = \min\left(\frac{L}{\Delta t}, \frac{\alpha L}{\Delta t - \hat{w}}\right) \tag{6}$$

Then, the aggregate amount of data stored in the proxy buffer and distributed among clients (which does not have to be requested from the central server) is equal to:

$$W = \min\left(\frac{L}{\Delta t}, \frac{\alpha L}{\Delta t - \hat{w}}\right) \Delta t \tag{7}$$

Fig.5 shows the amount of data available locally, W, as the function of the client arrival rate for five values of α: 1.0, 0.8, 0.6, 0.4 and 0.2. The video length is set to 3600 s and the client buffer is δ=30 s of the video. The values of W are expressed as a percentage of the video length. We observe that as

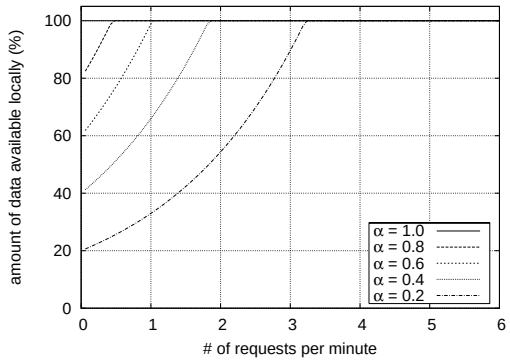


Fig. 5. Data available locally

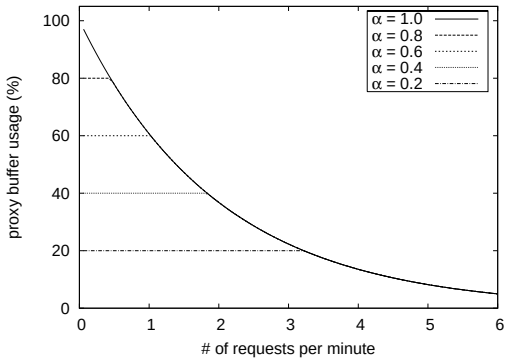


Fig. 6. Proxy buffer usage

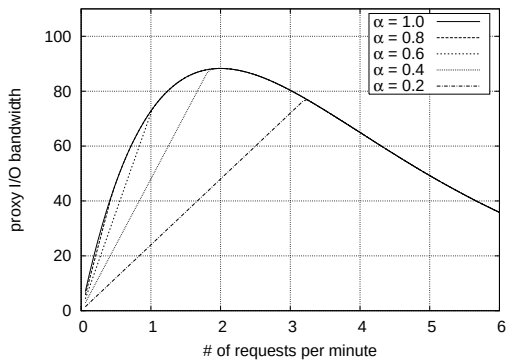


Fig. 7. Proxy I/O bandwidth

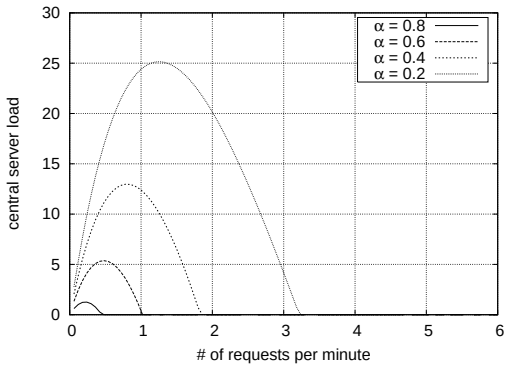


Fig. 8. Central server load

the arrival rate increases, the amount of data beyond prefix deposited at the proxy increases. Beyond a threshold arrival rate, the proxy has sufficient buffer space, due to the concurrent loops, for each loop to return all received data back to the proxy without discarding any frames. Eventually, the total amount of data the proxy needs to cache becomes smaller than the available storage space and the system becomes independent of the central server, i.e., all video frames are available either at the proxy or at client buffers. For example, if the proxy can cache 20% of the video (12 minutes), the entire video is available locally for the arrival rate of a little over 3 requests per minute. Beyond this point, the performance of the system is the same as that of the system with unlimited proxy buffer.

Given client arrival rate λ, we can identify the minimum amount of data that the proxy needs to cache to make an entire video available locally. The amount of data buffered by the clients is equal to B given by Eq. 2. Thus, the proxy needs to cache the remaining amount $L - B$. Let α_m denote the minimum amount proxy buffer space needed for an entire video to be available locally (expressed as a percentage of the video length). Then:

$$\alpha_m = 1 - \frac{B}{L} \tag{8}$$

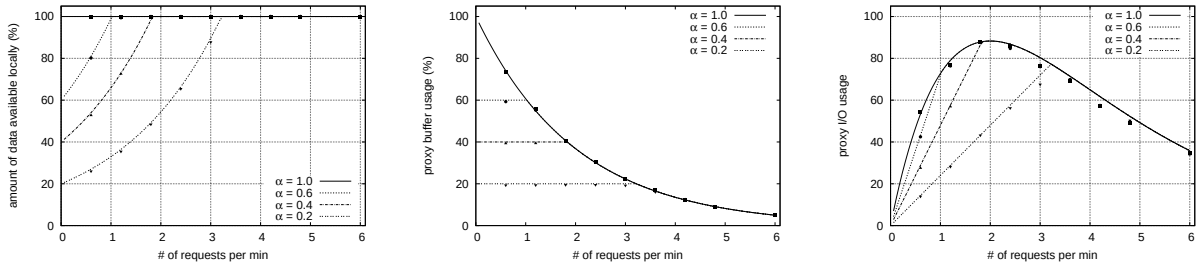
Naturally α_m decreases with an increase in the arrival rate since clients provide more and more buffer space.

C. Resource Usage Analysis

We now analyze proxy resource usage and the central server load as the functions of client arrival rate λ and available proxy buffer space αL. The proxy buffer utilization remains at 100% for a range of low arrival rates. In other words, the proxy caches αL of data as presented in Fig. 6. As the client arrival rate λ increases, the locally available data increases and the proxy buffer utilization decreases. Recall that the minimum amount of data the proxy needs to cache, α_m, is a function of λ. When α_m decreases to α as λ increases, the entire video is available locally. The higher the λ is, the lower the α_m is. The amount of data that the proxy needs to cache decreases with the increase of λ and eventually approaches 0. The resource usage presented in Figs. 6, 7 and 8 show values corresponding to the behavior presented in Fig. 5. That is, for video length L=3600 s, client buffer size δ= 30 s and five different values of α = 1.0, 0.8, 0.6, 0.4 and 0.2. The required proxy I/O bandwidth and central server load are presented with the video playback rate set as a unit.

The proxy I/O bandwidth usage is proportional to the number of loops receiving and returning data to the proxy, i.e, the approximate I/O rate is $2k$. Fig.7 shows I/O bandwidth usage for various values of α including the case when the proxy can cache the entire video (unlimited proxy buffer space) for the comparison. Since the proxy initially buffers a smaller amount of data in the limited buffer case, the I/O bandwidth usage is lower than that with the unlimited buffer case. As the arrival rate increases, the amount cached in the limited buffer case initially remains fixed (αL) but the cached data is divided into a larger number of chunks due to the increased number of loops. Consequently the I/O bandwidth usage increases but still remains lower than that in the unlimited buffer case. For the arrival rate large enough to make $\alpha_m = \alpha$, the amount of data cached by the proxy in both, unlimited and limited proxy buffer cases, is the same. From this point on, the amount cached by both systems remains the same and decreases with an increase in the arrival rate. The I/O bandwidth usage is consequently the same in both systems. We observe in Fig. 7 that the curves for $\alpha = 1.0$ and $\alpha < 1.0$ merge at the point where $\alpha_m = \alpha$. The smaller the value of α, the later (for a higher arrival rate) the merging takes place.

The central server load is proportional to the number of loops receiving data from the central server: $(L\lambda - 1)e^{-\lambda\delta} + 1 - k$. Recall that $(L\lambda - 1)e^{-\lambda\delta} + 1$ is the total number of loops and k is the number of loops that need no data from the central server. Fig. 8 presents central server load as a function of client arrival rate λ. We see that the load initially increases with the arrival rate and then decreases. The number of loops requesting suffix from the central server initially increases with the arrival rate. However, more



(a) Data available locally (b) Proxy buffer usage (c) Proxy I/O usage

Fig. 9. Model evaluation

loops no longer need data from central server as the arrival rate increases further and the requested suffix size decreases with the arrival rate increases. The number of loops also eventually decreases with larger arrival rates. Thus, the load reaches a maximum for a certain arrival rate and starts to decrease as the rate increases further. The larger the amount cached by the proxy, the lower the server load peaks and the earlier (for a lower arrival rate) it occurs. The server load approaches 0 when the entire video becomes available locally, i.e., for a λ that makes $\alpha = \alpha_m$.

D. Analysis Validation

In order to evaluate the analysis of the above model, we conducted a number of simulations. The length of a video in these simulations is set to 3600 s and the proxy is capable of caching α= 0.2, 0.4, 0.6 and 1.0 times the video length. The client buffer size is set to 30 s. Figure 9 shows the simulation results collected for a number of request arrival rates. These results are presented as dots in the Figure. The analytical results obtained with the model presented in this section are shown with different types of continuous lines in the Figure as the functions of the request arrival rate. The comparisons between the simulation results and the analytical results are conducted for the proxy resource usage and the amount of data available locally. The simulation results are averaged over a period of 6 hours. Overall, the simulations data are very close to the obtained analytical results.

VI. Proxy Buffer Utilization for Multiple Videos

In order to evaluate the influence of *Loopback* on the overall resource usage, we consider a set of videos accessed from a community of users. We assume that the storage space and the I/O bandwidth available at the proxy are limited. Hence, we have to decide what portion of each video should be cached by the proxy so that the proxy resources are utilized in the most efficient way. The goal is to minimize the amount of data requested from the central server by the users under the proxy storage and I/O bandwidth availability constraints.

A choice is made for an amount of data of each video that should be cached. The portion of data, denoted as previously by α, may be zero indicating that no part of a video is cached. The amount may be also equal to 1 indicating that the entire video is cached by the proxy. Generally, $0 \leq \alpha \leq 1$. We assume that if some part of a video is cached by a proxy, all users receive that part from the proxy and do not request it from the central server. Thus, caching a portion of a video implies certain I/O bandwidth usage at the proxy. Next, we formulate and solve the problem of selecting a cached amount for each video in the proxy server without *Loopback* and with *Loopback*. The resource constraints are the same in both cases. Our goal is to compare the efficiency of both schemes.

A. Problem Formulation

We first formulate an optimization problem for the system without *Loopback*. The value of the optimization function expresses the amount of data requested from the central server and depends on two factors: the amount of data not cached by the proxy and the popularity of the video. Naturally, the smallest value of the function is reached when the entire video set is cached by the proxy. However, the proxy has limited I/O bandwidth and limited disk space. We consider the second constraint to be of lesser importance since the disk capability increases faster than I/O bandwidth. The problem is formally defined as follows:

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{N}(1-\alpha_i)L_i\lambda_i \\ \text{subject to} & 1)\ \sum_{i=1}^{N}\lambda_i\alpha_i L_i \leq P_p \\ & 2)\ \sum_{i=1}^{N}\alpha_i L_i \leq B_p \\ & 3)\ 0 \leq \alpha_i \leq 1 \end{array} \tag{9}$$

where i is the index of the video and N is the total number of videos, P_p denotes the available I/O bandwidth and B_p denotes the available disk space at the proxy, respectively. Variable α_i determines the size of the prefix of video i that should be cached. The problem formulated in such a way is a linear programming problem. Since it is not possible to cache the entire video set, intuitively, the most popular videos should have the largest prefixes cached. When storage space is the bottleneck, then the largest prefixes are indeed selected for the most popular videos. Such a choice allows us to utilize the available disk space in the most efficient way. However, when I/O bandwidth is the bottleneck, there are a number of possible solutions, each yielding the same minimal value of the optimization function. Choosing the largest prefixes for the most popular videos yields the most efficient disk-usage solution. On the other hand, there is an advantage in having at least a small prefix cached for each video if possible. Thus, there is a variety of possible choices in this case but we concentrate on the most efficient disk-usage solution.

Similarly we formulate an optimization problem for the system utilizing *Loopback*. In this case α determines the amount of buffer space used by the proxy for caching video. Recall that in *Loopback*

the proxy caches non-consecutive chunks of frames and the locally available prefix is cached partially by the proxy and partially by the clients. The total amount of data cached can be bigger than αL. The optimization problem is formulated as follows:

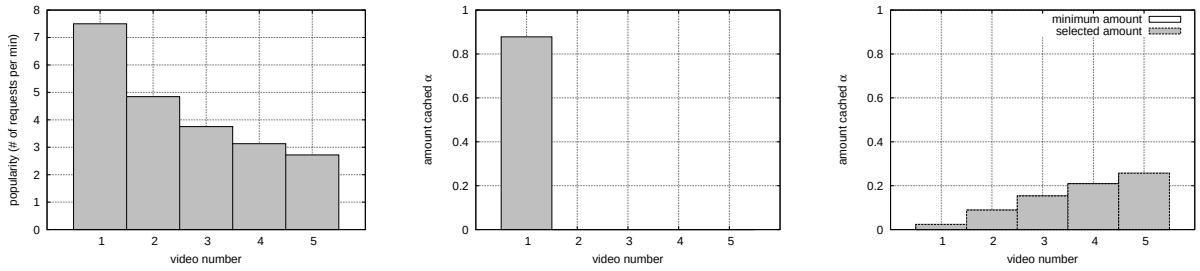
$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{N}(L_i\lambda_i - 1)e^{-\lambda_i\delta_i} + 1 - k_i \\ \text{subject to} & 1)\ \sum_{i=1}^{N}\alpha_i L_i \leq B_p \\ & 2)\ \sum_{i=1}^{N} 2k_i \leq P_p \\ & 3)\ 0 \leq \alpha_i \leq 1 \end{array} \tag{10}$$

where k_i is the number of loops receiving video i data only from the proxy and is given by Eq. 6. The value of the optimization function is expressed as a number of loops needed to receive data from the central server. This value for video i is equal to the difference between the total number of concurrent loops for this video given by $(L_i\lambda_i - 1)e^{-\lambda_i\delta_i} + 1$ and the number of loops receiving data only from the proxy k_i. The amount of data requested from the central server in this case is equal to the difference between the video length and the amount of data cached by the proxy and by the clients within the loops. The I/O bandwidth usage at the proxy is proportional to the number of loops receiving data from the proxy. Problem formulated in such a way is a non-linear optimization problem with non-linear constraints. After analyzing solutions obtained using numerical methods for various values of λ, we concluded that the amount of data cached and the space needed for the most popular videos is the minimum space required such that these videos are entirely locally available, i.e., $\alpha_i = \alpha_{m_i}$. Increasing the amount of data cached beyond α_{m_i} does not affect the I/O bandwidth usage at the proxy or the value of the optimization function but increases the required disk space usage. In other words there is no use caching more than α_{m_i} for a given video. Thus, the solution to the optimization problem can be obtained by choosing value α_m for as many videos as possible starting with the most popular ones, and as large amount as possible for the first video for which α_m cannot be cached.

B. Numerical Test Results

We evaluate the performance of *Loopback* mechanism with a set of videos through a number of numerical tests. In the first set of tests, we use a small video set. The set consists of five videos. We assume that the size of each video is 3600 s and the popularity is generated using Zipf distribution with the shape parameter 0.27 [1], i.e., $\lambda_i = \beta i^{1-0.27}$, where i is the rank of a video according to its popularity and β is a scaling factor. Each user buffers at most 30 s of a video ($\delta = 30$ s).

In the first example, the average number of requests per second for the most popular video is equal to 0.125 (roughly 7.5 requests per minute) with $\beta = 0.125$. The amount of I/O bandwidth available at the proxy is equal to 30% of the total amount needed to serve all videos from the proxy and I/O bandwidth at the proxy is the bottleneck.

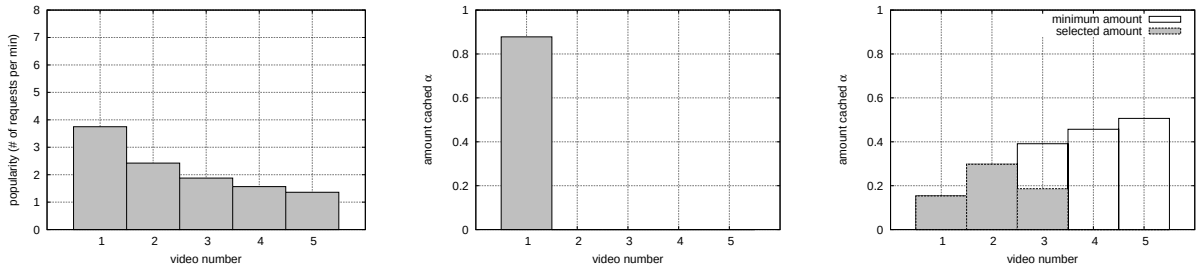


(a) video popularity (b) W/o loopback - minimum disk usage (c) With loopback

Fig. 10. Prefix choice for β=0.125 with I/O bottleneck

Figure 10(a) shows the popularity for five videos. Figure 10(b) presents the most efficient disk-usage of all optimal solutions for the prefix selection problem in the system without *Loopback*. Figure 10(c) presents the prefix choice made for each video in the system with *Loopback*. It is interesting to observe that the largest amount of data is selected for the least popular video. Figure 10(c) presents also, beside the amount selected, the minimum amount α_m needed for each video for the given popularity to make the entire video available locally. We observe that the largest amount is naturally needed for the least popular video. In fact in this example the amount of the available I/O bandwidth is sufficient enough to select α_m for each video. Thus, we see that the largest amount is selected for the least popular video.

In the second example, the amount of the resources available at the proxy is no longer sufficient to choose the minimum amount to make each video available locally in the case with *Loopback* mechanism. We achieve this effect by decreasing the popularity of the video set. The average number of requests per second for the most popular video is set to 0.0625 (roughly 3.75 requests per minute) with $\beta = 0.0625$. The popularity is presented in Figure 11(a). As in the first example, the amount of I/O bandwidth available still equals 30% of the total amount needed to serve all videos from the proxy without *Loopback*. We present the most efficient disk-usage prefix selection for the system without *Loopback* in Figure 11(b). We are able to cache about 90% of the most popular video. In the case with *Loopback*, due to insufficient amount of I/O bandwidth, the maximum amount α_m is selected only for the most popular videos. We see in Figure 11(c) that only the two most popular videos are entirely available locally. The amount cached for the third most popular video is smaller than the amount needed to achieve such an effect. The results obtained in these settings with a storage bottleneck are similar. Due to the space limitation, we do not present the results for the cases with storage as bottleneck.



(a) Video popularity (b) Without loopback (c) With loopback

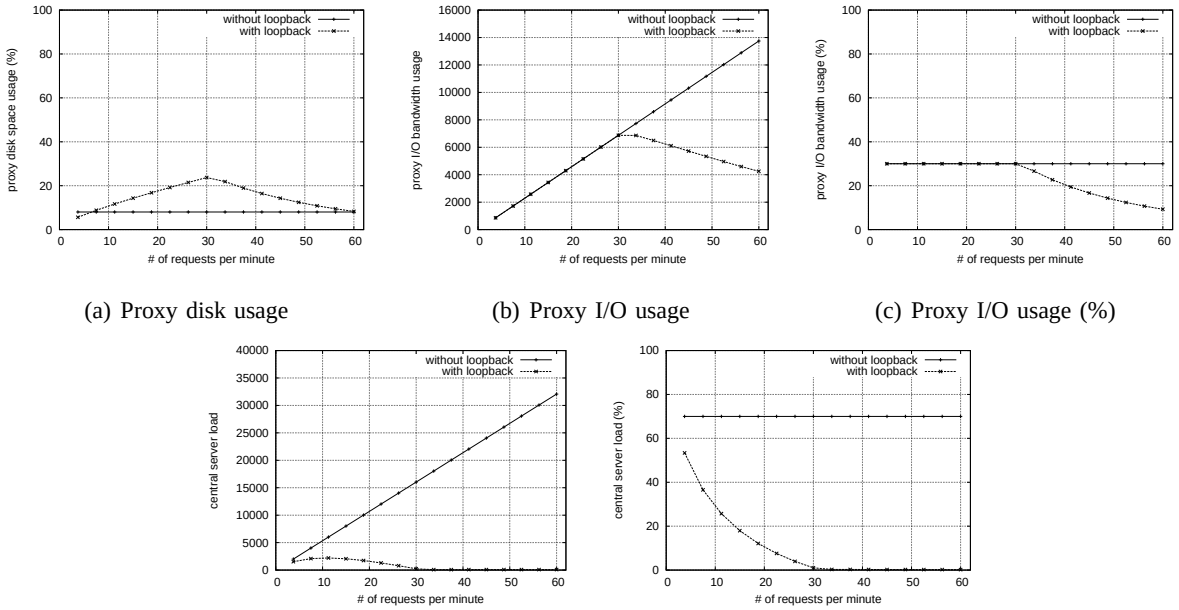
Fig. 11. Prefix choice for $\beta = 0.0625$ with I/O bandwidth bottleneck

C. Resource Usage Dependence on Popularity

Next we compare the resource usage at the proxy: I/O bandwidth and disk space, and the bandwidth at the central server in both systems. We increase the video set to 100 videos. Similarly to the previous set of tests, each video has length of 3600 s and the popularity is generated from the Zipf distribution. A user caches at most 30 s of a video. The range for the popularity scaling factor β is from $\frac{1}{32}$ to 1./

1) I/O Bandwidth Bottleneck: First, we set the I/O bandwidth available at the proxy equal to 30% of the amount needed to serve all requests from the proxy without *Loopback*. Note that as the video popularity increases, the required I/O bandwidth increases and is always kept at 30% level throughout the entire test.

Figure 12(a) presents the disk usage at the proxy as a percentage of the disk space needed to cache the entire video set. We use the most efficient disk-usage choice of prefixes in the system without *Loopback* in this comparison. We observe that in the system without *Loopback* the disk usage remains constant and is below 10%. Recall that as the video popularity increases, so does the required I/O bandwidth and the available I/O bandwidth is always equal to 30% of the required bandwidth. The disk space usage in the system with *Loopback* changes with the popularity. Initially, for low popularities, the disk usage is low. The space needed to make the entire video available locally is large and the amount of I/O bandwidth available allows to cache enough data to make only a small number of the most popular videos available locally. As the popularity increases the space needed for local availability decreases and the number of videos for which such an amount is cached increases. Thus, we observe an increase in the disk space usage. For some value of popularity ($\beta \approx 0.45$) the amount cached for each video ensures its local availability. From this point on, the disk space usage starts decreasing as the amount cached decreases. The disk usage with *Loopback* is lower than that without *Loopback* for low popularities when most loops contain only one client. Recall that the I/O bandwidth consumption at the proxy due to a loop is twice the I/O bandwidth consumption due to one client without *Loopback*. Thus, even if both systems use the



(a) Proxy disk usage (b) Proxy I/O usage (c) Proxy I/O usage (%)

(d) Central server load (e) Central server load (%)

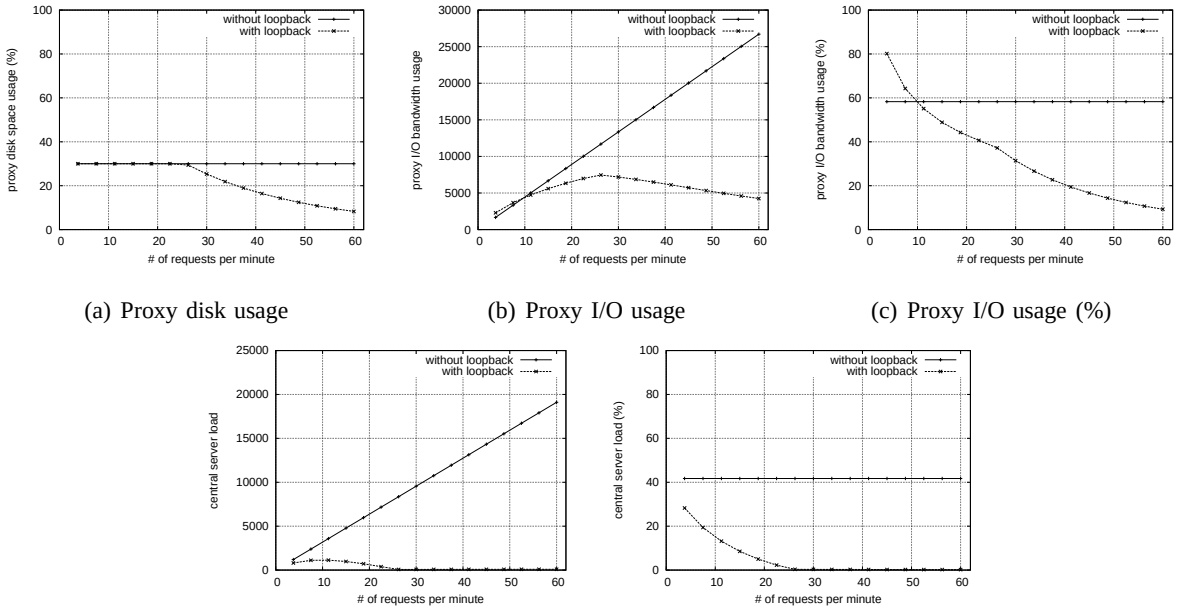
Fig. 12. Resource usage with I/O bandwidth bottleneck

same amount of I/O bandwidth, system with *Loopback* caches less.

Figure 12(b) shows the I/O bandwidth usage at the proxy. In the system without *Loopback*, the I/O bandwidth usage increases linearly with the popularity with the slope of 0.3. The behavior of the system with *Loopback* is initially similar. However, when the popularity increases beyond 0.45 and all videos are available locally, the system does not utilize the entire amount of I/O bandwidth. A smaller amount is sufficient to make the system independent of the central server. Figure 12(c) shows the same values as a percentage of the total amount needed to serve the entire video set from the proxy without *Loopback*.

Figure 12(d) shows the value of the optimization function, i.e., the central server load resulting from a given proxy caching choice as a function of video popularity. We observe that in the system without *Loopback*, the central server load increases with the popularity in a linear manner. In the system with *Loopback*, the load first slightly increases and then decreases reaching 0 for $\beta \approx 0.45$. At that point, all videos become available locally. Figure 12(e) presents the central server load as a fraction of the load we would observe without proxy caching. We see that in the system without *Loopback*, the ratio is constant and close to 0.7, while in the system with *Loopback*, the ratio decreases until it reaches 0.

We draw the following conclusion from this set of tests. Loopback allows us to reduce the central server load much more than the standard proxy caching mechanism without Loopback. The difference is larger for larger video popularities. Although Loopback uses generally more disk space, it needs less proxy I/O bandwidth at the same time. The results confirm also that Loopback becomes more and more



(a) Proxy disk usage (b) Proxy I/O usage (c) Proxy I/O usage (%)

(d) Central server load (e) Central server load (%)

Fig. 13. Resource usage with storage space bottleneck

efficient as the popularity increases.

2) Disk Space Bottleneck: In the next set of tests, the storage space is the bottleneck, i.e., the proxy can cache no more than 30% of the entire video set. Figure 13(a) presents the disk usage at the proxy as a function of popularity. We observe that the disk space usage in the system without *Loopback* is constant and equal to 30%. The disk space usage with *Loopback* starts at value 30% for low video popularity and then decreases. A decrease is caused by the fact that the amount cached for each video to ensure local availability keeps decreasing with increasing popularity.

Figure 13(b) shows the I/O bandwidth usage at the proxy in both systems. The usage increases linearly with the popularity in the system without *Loopback*. It remains close to 60% of the amount needed to serve the entire video set from the proxy. The I/O bandwidth usage at the proxy in the system with *Loopback* is initially larger than that without *Loopback*. Recall that for low popularity most loops contain only one client and consume more bandwidth than one client in a *non-Loopback* system. The I/O bandwidth usage initially increases with the popularity. Note that the proxy caches the same amount of data but the popularity of this data increases. For some value of disk space usage, the proxy is able to cache enough data for the system to become independent of the central server. From this point on, the I/O bandwidth usage decreases since the amount of data cached and the number of client loops decreases.

The central server load is lower than in the previous set of tests but its behavior as a function of popularity is similar. The load keeps increasing linearly in the system without *Loopback* and its ratio to

maximum load is constant. In the system with *Loopback*, the central server load first slightly increases as the popularity increases and then starts decreasing.

We draw the following conclusions from this set of tests. Loopback is more efficient in using proxy resources, i.e., it uses less storage space and less I/O bandwidth. The central server load is much lower with Loopback than that with the traditional non-Loopback caching.

VII. Client Failure Influence on Loopback Performance

Since *Loopback* depends on client collaboration, we need to clearly understand how unreliable clients affect the effectiveness of the proposed scheme. We first describe the effect a client failure has on the playback of other clients receiving a given video and on the resource usage at the central server and the proxy. Next, we show how redundant caching increases the reliability of our scheme.

A. Failure Influence on Playback

When a client failure occurs, the data buffered by the client is lost. Recall that the proxy server does not keep frames that have been forwarded to clients. Thus, the lost data has to be obtained from the central server which incurs certain delay. Due to the nature of the client collaboration of *Loopback* such a failure may affect only clients succeeding the one that failed. The number of clients affected and the length of playback disruption experienced by these clients depend on the client arrival rate and the buffer size.

Let us assume that the i^{th} client in a loop failed. Then, the $(i+1)^{th}$ client experiences playback disruption of length equal to d_s, the time needed for the central server to deliver the lost data. However, this client still transmits data it has in its own buffer to the $(i+2)^{th}$ client. The $(i+2)^{th}$ client experiences disruption only if the time needed to transmit data cached by $(i+1)^{th}$ client is shorter than delay d_s. The disruption, if experienced by the $(i+2)^{th}$ client, lasts for $d_s - b_{i+1}$, where b_{i+1} is the buffer occupancy of the $(i+1)^{th}$ client. Similarly, the $(i+3)^{th}$ client experiences disruption only if the amount of data cached by the two preceding clients is not sufficient to play for a duration of d_s. Clients in the subsequent loops may be affected by the failure only if $d_s > \delta$. Typically, d_s is smaller than δ and we consider only the failure effect on the clients within the same loop.

Generally, the probability that the number of clients C affected by the failure within the same loop is equal to or larger than a number c, is equal to the probability that at least C consecutive clients arrived within a time interval of length d_s (for $d_s < \delta$):

$$P(C \geq c) = P(X_1 \leq \delta, X_2 + X_3 + ... + X_c < d_s) = (1 - e^{-\lambda\delta}) \left(1 - e^{-\lambda d_s} \sum_{i=0}^{c-1} \frac{(\lambda d_s)^i}{i!}\right) \qquad (11)$$

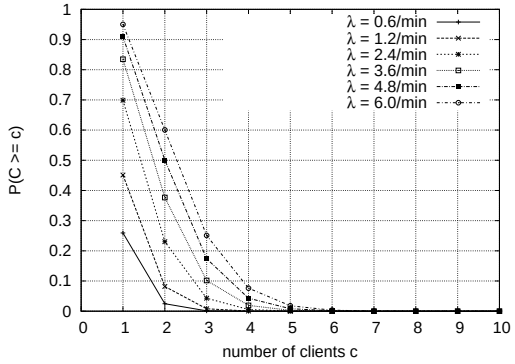


Fig. 14. Number of clients affected by failure

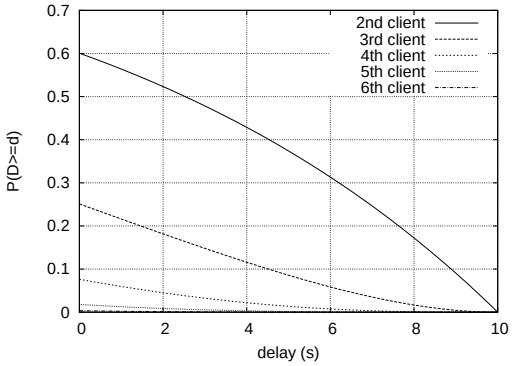


Fig. 15. Playback disruption due to failure

where X_i is a random variable representing the client inter-arrival time distributed exponentially. Fig. 14 shows the probability that the number of clients affected is larger than or equal to c for several different values of client arrival rate. In this example the clients buffer no more than 30 s of a video and d_s= 10 s. The number of clients affected is smaller for a lower arrival rate due to that fact that the client arrivals are more spread in time and each client buffers more data. For example, with the average of 6 requests arriving per minute, the probability that two or more clients are affected by a failure is close to 0.6, while for 0.6 requests per minute this probability is equal to only 0.025.

The probability that the length of disruption experienced by the $(i+c)^{th}$ client is no shorter than d is given by:

$$P(D \geq d) = P(X_1 \leq \delta, X_2 + ... + X_c \leq d_s - d) = (1 - e^{-\lambda\delta}) \left(1 - e^{-\lambda(d_s - d)} \sum_{i=0}^{c-1} \frac{(\lambda(d_s - d))^i}{i!} \right) \quad (12)$$

Fig. 15 shows $P(D \geq d)$ for a number of clients succeeding the client which failed. Recall that the disruption experienced by the first succeeding client last exactly d_s. The clients buffer no more than 30 s of a video and the arrival rate is 6 requests per minute. We observe that the second client experiences delay with probability close to 0.6. The probability that the delay for this client is larger than or equal to 3 s is equal to 0.48, for the third client the probability is equal to 0.15 and for the fourth client the probability is equal to 0.03. We observe that for lower arrival rates the delays are shorter. For example, the probability of a delay longer than or equal to 3 s for the arrival rate of 3 requests per minute is equal to 0.28 for the second client and 0.05 for the third client.

These observations show that the higher the demand for a video, the larger the influence of a failure on the performance. This effect of a failure can be limited with redundant caching. Fortunately, clients of a very popular video use only a small portion of their buffer space and the remaining part can be utilized to increase reliability. We show how to achieve such an increase in the next Section.

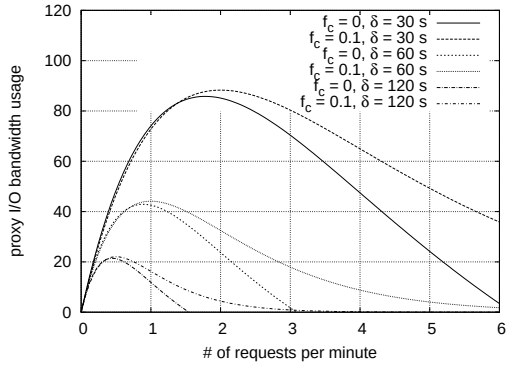


Fig. 16. Proxy I/O bandwidth usage

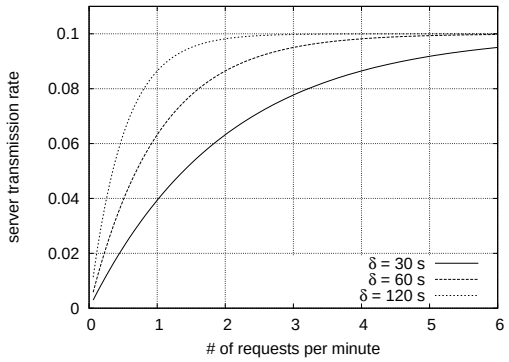


Fig. 17. Server data rate

B. Failure Influence on Resource Usage

We now evaluate the influence that a client failure has on the resource usage of the central server and of the proxy. An increase in the resource usage is caused by a failure of a client which has at least one successor in a loop. A failure causes a multi-client loop to be broken into two separate loops. If the i^{th} client fails, all clients preceding the i^{th} client in a loop form a separate loop with the $(i-1)^{th}$ client returning data to the proxy. Meanwhile, all clients after the i^{th} client form another separate loop with the $(i+1)^{th}$ client receiving data from the proxy. Hence, the proxy has to accommodate two more streams, one incoming and one outgoing. On the other hand, a failure in a one-client loop decreases the number of streams by two. The average I/O bandwidth usage at the proxy is computed as

$$\hat{P}_c = 2L\lambda e^{-\lambda\delta} - \frac{1}{2}\left(2L\lambda f_c e^{-2\lambda\delta} - 2L\lambda f_c(1-e^{-\lambda\delta})^2\right) = L\lambda(2e^{\lambda\delta} + f_c(2e^{-\lambda\delta}-1)) \qquad (13)$$

From the average I/O bandwidth usage given by Eq. 3, we subtract two streams due to the failure of one-client loops and add two streams created due to the client failures in multi-client loops. We assume that the client failure is equally likely to happen at any time of the client's lifetime. Therefore, the probability that the failure has already happened for a given client is on the average equal to $\frac{1}{2}$, and we use this factor to scale the number of clients which have failed. Fig. 16 illustrates the effect of failure probability on the proxy I/O bandwidth usage for three values of client buffer size: 30 s, 60 s and 120 s, and for failure probability equal to 0.1. For comparison purpose we show also proxy I/O bandwidth usage without failures. We observe that the increase of the proxy I/O bandwidth is more significant for high client arrival rates since most failures increase the number of loops by one in that region. However, the I/O bandwidth usage depends also strongly on the failure probability and for small values of f_c, the increase is not significant.

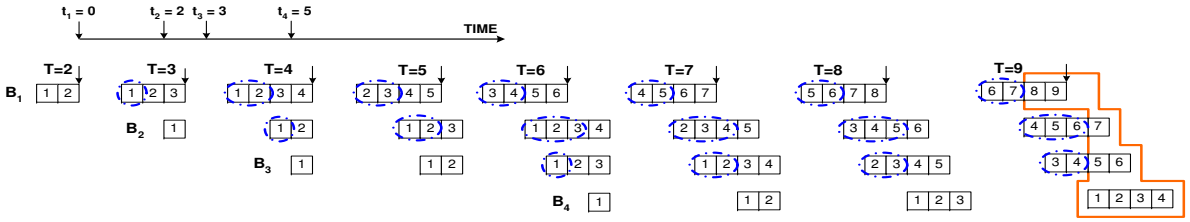


Fig. 18. Buffer immediately and fill a buffer to its maximum

Data lost due to client failures has to be obtained from the central server as a patching stream. We now estimate server's transmission rate due to failures. The average amount of data buffered by a client is equal to $\delta e^{-\lambda\delta} + \hat{a}(1 - e^{-\lambda\delta})$, where $\hat{a}$ is an average amount buffered by a client in a multi-client loop and is equal to $\frac{1-\lambda\delta e^{-\lambda\delta}-e^{-\lambda\delta}}{\lambda(1-e^{-\lambda\delta})}$. Given the client failure probability f_c, the amount of data requested from the server over a unit of time is equal to

$$\hat{R}_c = f_c\lambda(\delta e^{-\lambda\delta} + \hat{a}(1 - e^{-\lambda\delta})) = f_c\lambda\left(\delta e^{-\lambda\delta} + \frac{1 - e^{-\lambda\delta}(\lambda\delta + 1)}{\lambda}\right) \tag{14}$$

Fig. 17 shows server's average transmission rate (in the units of video playback rate) due to failures as a function of the client arrival rate for three values of the client buffer size. We observe that the transmission rate increases as the arrival rate increases and approaches a value equal to the failure probability.

VIII. Enhancing Performance and Reliability with Duplication

In the basic scheme discussed in the previous section, when a collaborative client fails, other clients may experience service disruptions and the server has to transmit extra data to repair such failures. To address these issues and further ensure service quality and survivability, we propose a simple *caching duplication* that exploits collaborative client buffers to cache multiple copies of data to deal with collaboration failures. This simple approach not only significantly reduces server load by locally repairing failures with client collaboration but also shortens the repairing delay caused by transmitting missing data from the central server in the basic repair scheme.

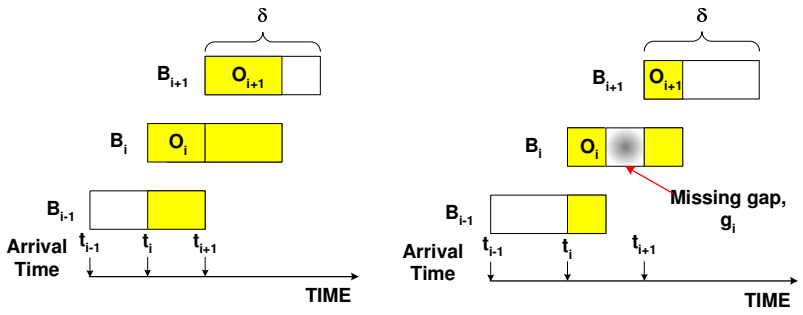
The basic scheme does not fully utilize the capacity of client buffers. A client starts to buffer and to play the video immediately at t_i so that there is no startup delay. A client stops to increase the amount of data buffered at $t_{i,e} = t_{i+1}$, where $t_{i,b} = t_i$ is the instant that a client starts to collect the data into its cache and $t_{i,e}$ is the instant that a client starts to replace the current buffer content with new data. Therefore, the buffer occupancy at client i is $b_i = t_{i+1} - t_i$. Since the mean distance between a pair of arrival is $1/\lambda$, we know that the average size of data buffered at a client is no more than $1/\lambda$, where we assume that the CBR video rate is 1. So the buffer utilization is only around $\frac{1}{\lambda\delta}$.

In order to improve the system performance and service availability, we propose a caching duplication that employs a *Buffer-immediately-to-maximum Policy*. Under this policy, a client starts to buffer data without any delay and buffers *as much as possible*, i.e., $t_{i,b} = t_i$, and $t_{i,e} = t_i + \delta$, where δ is the given client buffer size. As shown in Fig. 18, this policy allows the duplication of data across collaborative caches such that we can utilize these naturally duplicated data to fix single- (and even multiple-) client failures. In the example, we have four clients arriving at time 0, 2, 3, and 5, and each client has a buffer of four units. Without failures, for a loop with k clients, we have a continuous sequence of L_{loop} units, where $L_{loop} = t_{k,e} - t_{1,b} + 1$ units from unit $(t - L_{loop} + 1)$ to unit t, where t is the current time. L_{loop} can be expressed as $\delta + \hat{t}$ given in Section IV. At time 9, we have a complete sequence of unit 1 to unit 9 from $t_{4,b}$ to $t_{1,e}$. In addition, we have duplicate copies of these units, marked in ellipses. The amount of duplicate data is determined by the client arrival times. The total number of duplicate units is $k \cdot \delta - L_{loop}$. An important property of this approach is that the higher the client arrival rate (i.e., a video with higher popularity), the better fault-tolerance is provided.

A. Complete- and Partial- Local Repair of Single Failures

In the previous basic scheme, when a failure occurs at client i, i.e., when client i fails to pass data to the next client in the loop, we ask the server to send a patching stream of the missing data to client $(i+1)$. Therefore, every client downstream in a loop may experience a service disruption of d_s seconds. We consider this type of failures as single failures. Note that multiple single failures may occur at the same time as long as they are not consecutive in a loop. To avoid the data transmission from the central server, we let client $(i-1)$ directly pass the duplicated data to client $(i+1)$ in a loop so that only minimum amount of data needs to be delivered from the central server to client $(i+1)$ in a repair. We name this operation as a *local repair*.

Fig. 19 show the relations between the buffers at consecutive clients. The overlap between client i's buffer b_i and client $(i-1)$'s buffer b_{i-1} is denoted as o_i, where $o_i = t_{i-1} + \delta - t_i$. The overlap between client i's buffer b_i and client $(i+1)$'s buffer b_{i+1} is denoted as o_{i+1}, where $o_{i+1} = t_i + \delta - t_{i+1}$. When client i fails, its preceding client $(i-1)$ may be able to provide all or partial data that are supposed to be passed to client $(i+1)$ by client i in a loop. In particular, as shown in Fig. 19(a), when $t_{i+1} - t_{i-1} \leq \delta$, i.e., there is no gap between b_{i-1} and b_{i+1}, we are able to perform a *complete-local repair* to fix the failure of client i by asking client $(i-1)$ to pass data directly to client $(i+1)$. Otherwise, as shown in Fig. 19(b), when $t_{i+1} - t_{i-1} > \delta$, there is a gap g_i between b_{i-1} and b_{i+1} due to the failure of client i, where $g_i = \delta - o_i - o_{i+1} = t_{i+1} - (t_{i-1} + \delta)$. We then perform a *partial-local repair*, in which client



(a) Completely local repair (b) Partially local repair

Fig. 19. Repair of Single Client Failure

$(i+1)$ obtains a portion of missing data (shown as o_i) from client $(i-1)$ and the gap g_i from the central server.

In a complete-local repair, client $(i-1)$ provides all data that client $(i+1)$ needs, because there is no gap between their buffers, i.e., $t_{i+1} - t_{i-1} \leq \delta$. Given an exponential arrival rate with a mean of λ, the probability p_1 of a successful completely-local repair for a single failure is equal to the probability that the distance between the arrivals of client $(i-1)$ and client $(i+1)$ is no more than δ in a loop, given as follows.

$$p_1 = \quad P[X_1 + X_2 \leq \delta, X_1 \leq \delta, X_2 \leq \delta] = \quad 1 - e^{-\lambda\delta}(1+\lambda\delta) \tag{15}$$

In a partial-local repair, $\delta < t_{i+1} - t_{i-1}$, client $(i-1)$ contributes a portion of missing data (as o_i in Fig. 19(b)) to client $(i+1)$, and we need the central server to send the missing data (as gap g_i in Fig. 19(b)) to client $(i+1)$. Since client i and client $(i+1)$ are in the same loop, we also have $t_{i+1} - t_{i-1} \leq 2\delta$. Similar to the reasoning of Eq.15, we obtain the probability of a partial-local-repair as follows.

$$p_2 = P[\delta < X_1 + X_2 \leq 2\delta, X_1 \leq \delta, X_2 \leq \delta] = (e^{-\lambda\delta} - e^{-2\lambda\delta})(2e^{-\lambda\delta} - 1) + \lambda\delta(e^{-\lambda\delta} - 2e^{-2\lambda\delta}) \tag{16}$$

Fig. 20 shows the probabilities of successful complete- and partial local repairs. As the mean arrival rate increases, we are more likely to repair a single failure locally with a complete-local repair, shown as the middle curve. In the meantime, a partial local repair is less likely, shown as the bottom curve since the arrivals tend to overlap with each other as shown in fig. 19(a). The top curves show the total probability of local repairs. Clearly, we have a good chance to perform local repair and reduce most repair-related load at the central server as presented in the following.

Using local repairs, we can significantly reduce repair-related overheads on the central server(s) when the request arrival rate is moderately high. Without a local repair, the central server needs to deliver l_{basic}

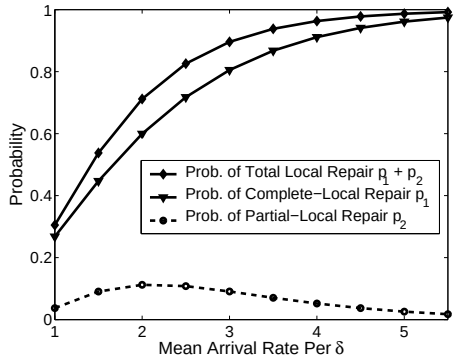


Fig. 20. Probability of Local Repair

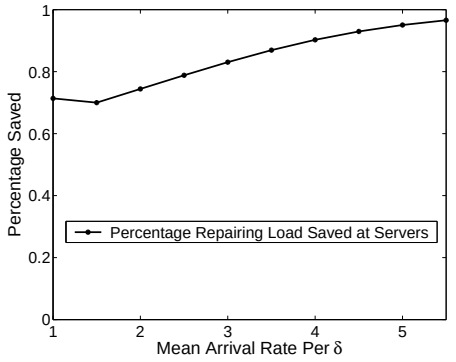


Fig. 21. Additional loads saved by local repairs

data to client $(i+1)$ for a single failure, where $l_{basic} = o_i \cdot p_1 + (o_i + g_i) \cdot p_2$. With a local repair, the server only needs to deliver l_{repair} data to client $(i+1)$ for a single failure, $l_{repair} = g_i \cdot p_2$. Clearly, $l_{repair}/l_{basic} \leq 1$. We denote this ratio as p_{saved}, representing the percentage of additional repair-related load saved by exploiting local repairs.

$$p_{saved} = \frac{E(o_i \cdot (p_1 + p_2))}{E(o_i \cdot p_1 + (o_i + g_i) \cdot p_2)} = \frac{1}{1 + \frac{(\delta - 2\hat{a}) \cdot p_2}{(\delta - \hat{a}) \cdot (p_1 + p_2)}} \tag{17}$$

where $\hat{a}$ is the mean distance between clients in a multi-client loop equal to $\frac{\lambda\delta e^{-\lambda\delta} + e^{-\lambda\delta} - 1}{\lambda(e^{-\lambda\delta} - 1)}$.

Fig. 21 shows that we can eliminate most repair-related load at the central server. Especially, when the arrival is moderately high, e.g., three arrivals per δ, we can save more than 80% of repair-related load.

B. Complete- and Partial-Local-Repair of Multiple Failures

We can easily extend the above model for single failures to multiple consecutive failures. In this case, multiple consecutive clients in a loop fail together. When the client arrival rate is high or the client buffer size is large, we have more duplicate units that form a continuous subsequence of the data held by a loop. Due to space limitation, we only present the complete local repair for multiple consecutive failure case in this paper. Other cases can also be obtained through similar procedures as the above. In general, when the distance between client $(i-m)$ and client $(i+1)$ is not more than δ, we can fix a burst of consecutive m client failures (including client $i-m+1$ to client i) with a probability as

$$p_m = P[X_1 + X_2 + \cdots + X_m \leq \delta, X_1 \leq \delta, X_2 \leq \delta, \cdots, X_{m-1} \leq \delta] = \frac{1 - e^{-\lambda\delta}(1 + \frac{\lambda(\lambda\delta)^{m-1}}{(m-1)!})}{\lambda} \tag{18}$$

C. Simulation Evaluation of Caching Duplication

To confirm the above analysis, we also use simulations to evaluate the proposed local-repair approaches. In these simulations, we choose the given client buffer size δ as 2% of the video length, and we examine

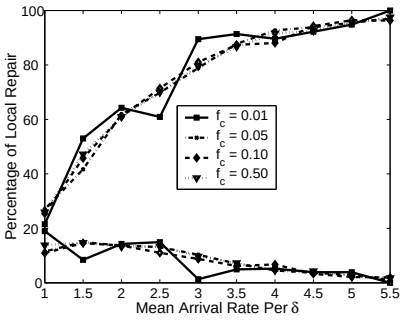


Fig. 22. Local repair probability

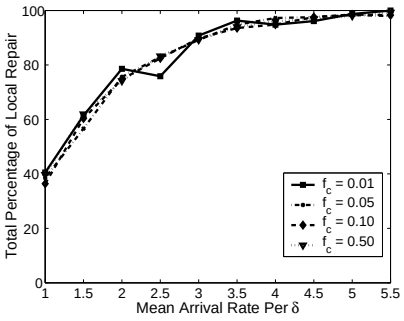


Fig. 23. Local repair percentage

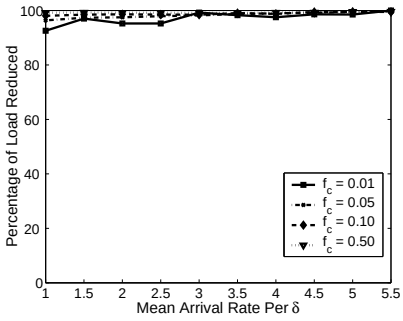


Fig. 24. Repair-related load saving

different arrival rates relative to δ, i.e, one arrival per δ to five arrivals per δ. Furthermore, we use four different client failure probabilities, i.e., $f_c = 1\%$, 5%, 10%, and 50%, in our test for covering a broad failure range. The length of simulation is over 10 times of the video length.

Fig. 22 shows the probabilities of complete- and partial-local repairs. The upper group of four curves are the probability of complete-local repairs under the four client failure probabilities. As we can see, the trends of these curves are very close to the middle curve in Fig. 20, which further confirms our analysis. The lower group of four curves are the probabilities of partial local repairs under the four client failure probabilities. The trends of these curves are very close to the bottom curve in Fig. 20. Fig. 23 shows the total probabilities of all local repairs under the four client failure probabilities. The trends of these curves are very close to the top curve in Fig. 20.

Fig. 24 shows the actual percentages of repair-related load measured in our simulation. Clearly, for all different arrival rates and client failure probabilities, local repairs are able to save more than 90% of repair-related load used in the basic scheme.

IX. Conclusions and Future Work

We have introduced a *loopback* mechanism for exploiting client collaboration in a two-level video streaming architecture. We have analyzed the resource usage of the proposed mechanism, namely, server I/O and network bandwidth, proxy I/O and network bandwidth, and proxy storage space. We have shown that even with limited client resources, *Loopback* can significantly reduce the requirements of network bandwidth, I/O bandwidth and buffer space at proxy servers, for moderately and highly popular videos. We have also analyzed the effect of client failures and developed local repair approaches to minimize playback disruption and the costs of repairs. Finally, we have shown that, even with limited proxy buffer space, the *Loopback* scheme is still able to significantly reduce the resource requirements at proxy servers.

We are investigating the broad enhancement of the basic *Loopback* model to allow varying amount

of resources committed by each client. In other words, we will consider the case when each client may specify how much disk space can be utilized, how many clients each client is willing to serve, and for what period of time.

REFERENCES

[1] S. Acharya, B. Smith, and P. Parnes. Characterizing user acess to videos on the World Wide Web. In *Proceedings of Multimedia Conferencing and Networking*, 2000.

[2] C. Aggarwal, J. Wolf, and P. Yu. On optimal batching policies for video-on-demand storage servers. In *Proc. of International Conference on Multimedia Systems'96*, 1996.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM*, pages 205 – 217, 2002.

[4] Y. Chu, S. Rao, S. Seshan, and H. Zhang. A case of end system multicast. *in Proc. of ACM SIGMETRICS 2000*, 2000.

[5] Y. Dong, E. Kusmierek, Z. Duan, and D. Du. A hybrid client-assisted streaming architecture: Modeling and analysis. *in Proc. of* 8_{th} *Internet and Multimedia Systems and Applications IMSA*, Aug., 2004.

[6] M. Guo and M. Ammar. Scalable live video streaming to cooperative clients using time shifting and video patching. In *Proc. of INFOCOM*, volume 3, pages 1501–1511, March 2004.

[7] Y. Guo, Y. Suh, J. Kurose, and D. Towsley. P2cast: Peer-to-peer patching scheme for VoD service. *in Proc. of WWW 2003, May 20-24, Budapest, Hungary*, 2003.

[8] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. PROMISE: Peer-to-peer media streaming using CollectCast. In *ACM Multimedia*, pages 45–54, November 2003.

[9] A. Hu. Video-on-demand broadcasting protocols: a comprehensive study. In *Proc. of INFOCOM*, pages 508–517, 2001.

[10] K. Hua, Y. Cai, and S. Sheu. Patching : A multicast technique for true video-on-demand services. In *Proc. of ACM Multimedia*, pages 191–200, 1998.

[11] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of USENIX OSDI*, pages 197–212, 2000.

[12] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proc. of the International World Wide Web Conference*, pages 252–262, 2002.

[13] W. Ma and D. H. C. Du. Reducing bandwidth requirement for delivering video over wide area networks with proxy server. *IEEE Transactions on Multimedia*, 4(4):539–550, December 2002.

[14] W. Ma and D. H. C. Du. Design a progressive video caching policy for video proxy servers. *IEEE Transactions of Multimedia*, 6(4):599–610, August 2004.

[15] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. *in Proc. of NOSSDAV*, pages 177–186, 2002.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[18] S. Sheu, K. A. Hua, and W. Tavanapong. Chaining: a generalized batching technique for video-on-demand systems. In *Proceedings IEEE International Conference on Multimedia Computing and Systems*, pages 110–117, 1997.

[19] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *1st International Peer To Peer Systems Workshop (IPTPS 2002)*, USA, Mar. 2002.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, 2001.

[21] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Proc. of IEEE INFOCOM*, volume 2, pages 1283–1292, 2003.

[22] D. Xu, H. Chai, C. Rosenberg, and S. Kulkarni. Analysis of a hybrid architecture for cost-effective streaming media distribution. In *SPIE/ACM Conf. on Multimedia Computing and Networking*, 2003.