Design and Implementation of a Caching System for Streaming Media over the Internet

Ethendranath Bommaiah, Katherine Guo, Markus Hofmann, Sanjoy Paul
Bell Laboratories, Holmdel, NJ 07733, USA
(ethen, kguo, hofmann, sanjoy)@dnrc.bell-labs.com

Abstract

Congested networks and overloaded servers resulting from the ever growing number of Internet users contribute to the lack of good quality video streaming over the Internet. We propose a design and implementation of a caching system for streaming media, which utilizes its local memory and disk resources to reduce network and server load, while also improving the video and audio quality perceived by end users. In particular, request aggregation, prefix caching and rate control mechanisms are used in the system design. The effectiveness of the system is validated through performance results from prototype implementation. As expected, the caching system reduces network and server load and improves client start-up latency.

1 Introduction

Streaming media delivery is gaining popularity as indicated by dramatically increased deployment of commercial products for playback of stored video and audio over the Internet [9], and proliferation of server sites that support audio/video content. Media quality as perceived by the end user however, is still very poor because support is lacking in the Internet to meet delay and jitter requirements for real time traffic. In particular, start-up latency and frenquency and length of interrupts of media streams increase significantly during periods of network congestion and media server overload. A number of techniques can be used to alleviate this situation.

Advanced encoding/decoding and smoothing mechanisms reduce the load on the network but not the load on the server. Support for quality of service in routers can improve quality by reducing delay, jitter and loss. However, we believe that application level caching entities called *helpers* placed within the network can not only provide benefits of transcoding and smoothing, but also reduce network load and server load by aggregating client requests as well as serving clients from a closer point in the network than the media host server.

Compared with traditional proxy caching for Web pages, streaming media presents new challenges. The real time requirements of multimedia streaming contributes to much of the complexity at the helper. Many of the media server functionalities such as scheduling and resource management need to be incorporated in the helper design. Given the large size of multimedia files and limited disk space at each helper, a limited number of files can be cached in their entirety.

This paper focuses on the design and implementation issues related to disk and memory utilization at a helper that inter-operates with media clients and servers using RTSP [13] and RTP [12] as their control and data protocols, respectively. This work demonstrates the advantage and feasibility of implementing a caching system for streaming media within the current Internet framework. The prototype implementation is used in validating the performance claims.

The rest of the paper is organized as follows. Section 2 presents the motivation and design of helpers in the network. Section 3 describes the implementation of a single helper. Section 4 discusses the performance evaluation followed by related work in Section 5. Finally, the conclusion is presented in Section 6.

2 The streaming cache design

The core elements of our streaming cache design are *helpers*, which are caching and data forwarding proxies inside enterprise networks or ISP networks. Helpers serve requests for streaming objects by sharing common resources as often as possible. Each client is associated with one helper which handles all the requests from the client. A client interested in getting a certain streaming object simply sends its request to the host server, but the request is redirected to the client's helper (H in Figure 1) either transparently by a layer-4 switch or by configuring the proxy in the client's software.

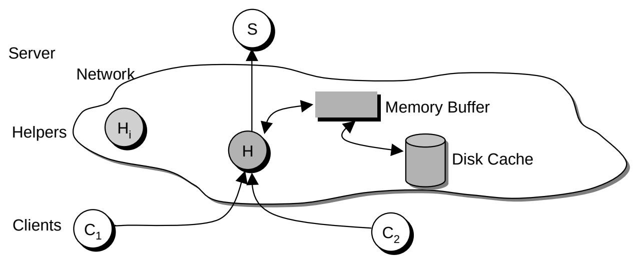


Figure 1. Application layer aware helper in the network.

On receiving the request, helper H serves the request itself if possible, otherwise it forwards the request to the most appropriate helper or the server. Each helper has limited disk, memory, network, and computational resources, and to maximize the number of accepted requests, it must make judicious decisions on how these resources are allocated and managed. The caching system consists of a network of helpers. Issues related to how these helpers cooperate, how to find the best helper and performance evaluations of the multi-helper system are persented in [7]. For clarity, this paper focuses on the design, operation, and evaluation of a single helper. This paper also does not discuss intelligent scheduling algorithms which can be adopted towards better utilization of computational resources.

The following sections discuss several techniques used in the helper design to better support streaming media over the Internet. These include *Segmentation of Streaming Objects* into smaller units for caching purposes, *Client Request Aggregation* using memory and disk resources at the helper, and *Data Transfer Rate Control* from the helper to fill the initial buffer at the client to reduce start-up latency.

2.1 Segmentation of streaming objects

There are two major differences between streaming objects and regular static objects like text and images on the web: their size and timing requirements. The size of streaming objects is normally an order or two larger than that of regular static web objects. For example, a single, two-hour long MPEG-I movie requires about 1.4 Gbytes of disk space. Given finite disk capacity, only a limited number of movies can be stored at a helper, which decreases hit probability and efficiency of the caching system. For example, storing a frequently accessed movie in its entirety at the helper makes a lot of sense. Storing an entire movie that will probably not be accessed in the future wastes disk space. It would be natural to divide movies into segments and distribute the segments among the helpers. However, it still makes sense to store very hot movies in their entirety.

Timing is the second fundamental difference between segments of a streaming object and a set of static objects, such as HTML pages or images. Each stream segment has a starting playback time and an ending playback time by which they are inter-related. Thus, when a streaming request arrives at a cache, it is not simply the question of a hit or miss. It is possible that the request will be a partial hit in the sense that one part of the requested streaming object is in the cache and the remaining parts are stored elsewhere. This is different from classical web caching. As a result, a requesting host might end up getting multiple pieces of the streaming object from different helpers or the media server. This not only increases signaling cost, but also increases the probability of losing synchronization. To reduce these costs, it is preferable to cache successive segments at a given helper rather than cache a sequence of segments with multiple gaps. This can be achieved by *prefix caching* [14] where segments of a movie are ejected from the end by replacing the last segment first. We record the last access time of each clip, and replace the Least Recently Used (LRU) clip from its end when new space is needed on disk[1]. This anticipates that users will play media objects from the beginning.

2.2 Client request aggregation

Requests for streaming objects are heterogeneous in nature. In particular, heterogeneity can appear in the following three forms:

- *Media object heterogeneity:* Different requests are for *different* media objects.
- *Arrival time heterogeneity:* Requests for the same media object tend to arrive at *different* time instances.
- *Range heterogeneity:* Each request for a media object is associated with a playback range. A request is often from the beginning of the object. However, VCR operations like rewind and fast forward will generate requests from the middle of the object.

Arrival time and range heterogeneity for the same media object can be represented by *temporal distance*. Consider two requests r_1 and r_2 for the same media object, where r_1 requests the object at time t_1 from p_1 seconds to its end, and r_2 requests it at time t_2 from p_2 seconds to its end, as shown in Figure 2. We define *temporal distance* between the two requests as the time difference between the requests for the same data packet, and this difference is $\Delta_{(r_1,r_2)} = (t_2 - p_2) - (t_1 - p_1) = (t_2 - t_1) - (p_2 - p_1)$.

A client request can be served out of memory or disk of its own helper, other helpers or the server, or any combination of them. The goal of a helper is to reduce the number

[1]Other cache replacement policies such as LFU can also be used.

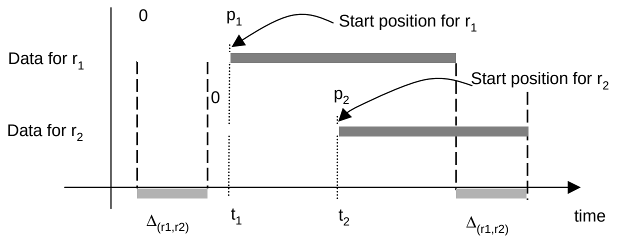
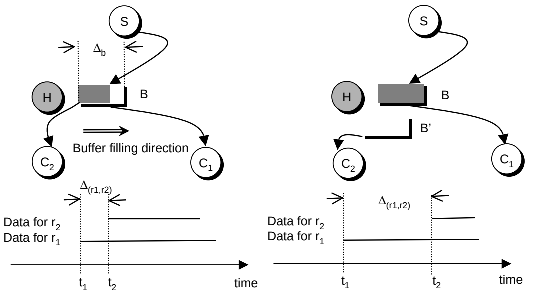


Figure 2. Temporal distance between two requests



(a)$\Delta_{r_1,r_2} \leq \Delta_b$ (b)$\Delta_{r_1,r_2} > b$

Figure 3. Ring buffer allocation for two requests r_1 and r_2.

of requests sent to the server, thereby reducing server load and network load. To accomplish this, the helper serves as many requests for the same object from its memory and disk as possible. This is called *client request aggregation* and is achieved by the buffering scheme described below.

A helper uses *ring buffers* in memory for short term storage and caches on disk for long term storage. A ring buffer of a given size can only serve requests arriving in a certain time range. The maximum temporal distance between two requests that can be served out of the same buffer is called *buffer temporal distance*.

The basic operation of a helper is similar to that of a regular cache. Whenever a request for a specific streaming object is received at a helper for the first time, the helper forwards the request to the server. Upon receipt of the request, the server starts streaming the object to the helper. The helper stores data in its memory and disk, and *at the same time* streams data to the client. In particular, the helper allocates a *ring buffer* in memory which corresponds to buffer temporal distance Δ_b seconds of data. The ring buffer operates as a moving window, and memory is reclaimed using a garbage collection scheme as discussed in Section 3.3.3. The ring buffer is allocated in anticipation of other clients requesting the same movie in the near future.

In the example represented in Figure 3, client C_1 requests a streaming object from its beginning at time t_1. This request r_1 is forwarded to the server by the helper H. Upon receiving data from the server, the helper H allocates a ring buffer B to store the first Δ_b seconds of the stream. At some later time t_2, client C_2 requests the same streaming object from its beginning via r_2. If t_2 is close enough to t_1, that is, their temporal distance $\Delta_{(r_1,r_2)} = t_2 - t_1 \leq \Delta_b$, then at time t_2, the beginning of the stream is still available from the ring buffer B[2]. Therefore r_2 can be served directly from the helper's buffer. This scenario is depicted in Figure 3(a).

If $\Delta_{(r_1,r_2)} > \Delta_b$ as shown in Figure 3(b), the beginning of the stream is not in the buffer B. any more. Instead, the helper allocates another ring buffer B' of size Δ_b seconds, and fills it with the beginning of the stream from either local disk, other helpers, or the server. At the same time, the helper streams data out of B' to client C_2.

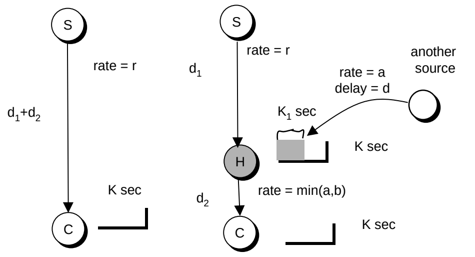
One objective of the ring buffer is to serve multiple requests with only one stream from the server. However, the ring buffer size is limited by memory resources which are scarce compared to disk space. Disk space is also limited at a helper. Therefore a streaming object might not be stored on disk at one helper in its entirety. As a result, a helper can serve a client's request from any combination of the following sources: ring buffer in its memory, cache on its disk, memory or disk of other helpers and the server.

2.3 Data transfer rate control

We follow the design principle that helper operation is transparent to the original client and server operation. After receiving a client's request, the media server sends out data packets according to its playback rate r bytes/second. Each client keeps a playout buffer of K seconds, and does not start playing the object until its playout buffer is filled. The purpose of this buffer is to absorb network jitter. When a client request is served from a helper at rate r, the initial playout buffer can be filled faster than from the server because of smaller network distance and reduced network congestion. Furthermore, the helper has the freedom to download data to the client as fast as possible to reduce client start-up latency even further. The mechanism for data transfer rate control is illustrated as follows.

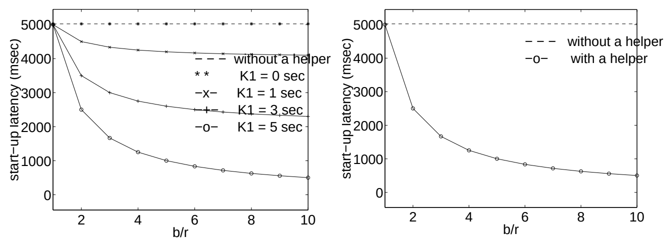
Figure 4 compares two scenarios: one involves a server and a client, the other involves a helper between the server and the client. We assume the delay between the server and the helper is d_1, and between the helper and the client is d_2. For simplicity, the delay between the server and the client is $d_1 + d_2$. The server sends streaming data at rate r bytes/second to the client or the helper depending on the scenario. One goal of the helper is to reduce client *start-up latency* which is defined as the time difference between

[2]For simplicity, we do not consider network delay in this formula.



(a) client-server (b) client-helper-server

Figure 4. Streaming rate between the helper and the client.



(a) from the server (b) from helper's local disk

Figure 5. Start-up latency when getting data from different sources.

sending a request and starting to play the media object at the client. In Figure 4(a), without a helper, the start-up latency is

$$L_0 = 2(d_1 + d_2) + K \tag{1}$$

where the factor 2 represents the round-trip time.

In Figure 4(b), assume the helper has K_1 seconds of data in its buffer where $0 \le K_1 \le K$. Helper H's buffer size is independent of K, however, only the initial K seconds are related to client's start-up latency. It takes d_2 seconds for the request to arrive at the helper. Once the helper receives the request, it starts two processes concurrently. One is to download the existing K_1 seconds of data to the client as fast as the bandwidth allows. Assuming the average rate between the helper and the client is b bytes/second, it takes $(K_1 r)/b$ seconds to download K_1 seconds of data. The other process is to request $K - K_1$ seconds of data from either its local disk, or another helper, or the server. Let d designate the one-way latency between these sources and the helper and let a designate the bandwidth from these sources to the helper. It takes $2d$ seconds for the first byte of the data to arrive at the helper. Thus the time for both processes to finish is $max(K_1 r/b, 2d)$.

Only after both processes finish, can the helper start sending the remaining $K - K_1$ seconds of data to the client. During this step, the buffer at the helper is filled with rate a and drained with average rate of b. In order to avoid buffer underflow, the actual draining rate for the buffer is set to $min(a, b)$, therefore the time for this part of data to arrive at the client is $d_2 + (K - K_1)r/min(a, b)$. The resulting start-up latency is then

$$\begin{aligned} L_1 &= d_2 + max(K_1 r/b, 2d) \\ &\quad + d_2 + (K - K_1)r/min(a, b) \end{aligned} \tag{2}$$

Consider the situation where the helper does not have any data cached, and compared with other helpers, the server is the best choice to get data. In this case, $K_1 = 0$, $d = d_1$, $a = r$, and $min(a, b) = min(r, b) = r$, therefore the start-up latency $L_1 = 2d_1 + 2d_2 + K$ which is the same as L_0.

The only parameter that the helper can control to improve start-up latency is the data transfer rate from the helper to the client. In the following example where the helper is requesting $K - K_1$ seconds of data from the server, $a = r$, and we set $d_1 = 10$ms, $d_2 = 1$ms, $K = 5$s. We vary the ratio between helper's data transfer rate b and the server's playback rate r between 1 and 10, and vary the amount of cached data K_1 between 0 and 5 seconds. When $K_1 = 0$, the latency is the same as getting data directly from the server. When $K_1 \ge 1$ second, K_1 seconds of data are transferred with rate b, and $K - K_1$ seconds of data are streamed with rate r. As shown in Figure 5(a), with a fixed K_1, the start-up latency decreases as b/r increases, and with a fixed b/r, the latency decreases as K_1 increases.

In the case where the remaining $K - K_1$ seconds of data are stored on the helper's local disk, we have $d = 0$, and disk bandwidth is greater than network bandwidth ($a > b$). Then the entire K seconds of data are transferred with rate b regardless of the value of K_1. The result is shown in Figure 5(b).

3 Implementation

This section discusses the implementation details of a helper. Although the design is fairly flexible to accommodate control and data protocols used by different vendors, we have implemented our helper using RTSP [13] as the control setup protocol and RTP [12] as the data transport protocol.

3.1 Overview

As the helper is located in the network, it behaves as a media server to the clients and as a media client to the servers. The helper also manages its local memory and disk resources. Figure 6 presents the main modules of a helper:

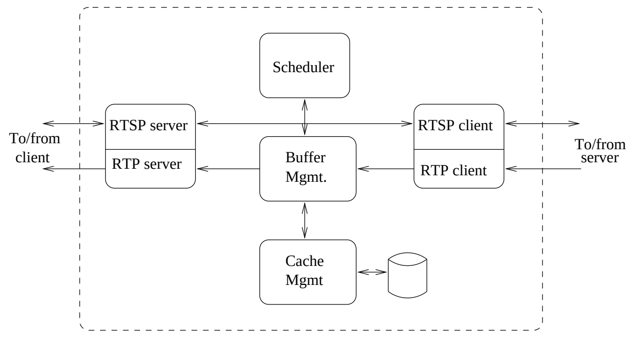


Figure 6. Main modules of a helper.

- **RTSP/RTP client and server:** The RTSP/RTP server module receives and processes RTSP requests from the clients, and interacts with the RTSP/RTP client module to forward them to the server after appropriate header translation; it also streams data to the clients using RTP. The RTSP/RTP client module contacts media servers or other helpers across the network to fetch data for client requests.

- **Buffer Management:** This module manages the available memory in the form of a pool of buffers, each associated with a media object, identified by an URL and a time range. It attaches a new incoming request to an existing buffer in the pool if possible; otherwise, it allocates a new buffer for the request as discussed in Section 2.2. It utilizes the RTSP/RTP client module to fetch data that is not available in local disk cache. It interacts with the Cache Management module when recording data for a media object into the cache and reading data for a media object from the cache.

- **Cache Management:** It maps URLs to local filenames and manages the disk space allocated for caching by implementing a suitable cache replacement policy (for example, LRU). It allows non-overlapping time segments of a media object to be recorded into and read from a single file.

- **Scheduler:** It manages the global queue of events, each scheduled to execute at a certain time. Example events include data producer and consumer events, and garbage collector events as discussed in Sections 3.3.2 and 3.3.3.

3.2 RTSP/RTP client and server

Unlike HTTP, the control protocol (RTSP, for example) used for media streaming is not a stateless protocol. Without a helper, the necessary state is managed by the media server. In the presence of a helper, the state for media streamed to the clients by the helper is managed by the helper. For example, in the case of RTSP, the session identifier will be issued and managed by the helper for sessions for which the helper is the data source. Note that this does not preclude the actual server from managing session state as helpers or end clients might still go directly to the server for the media.

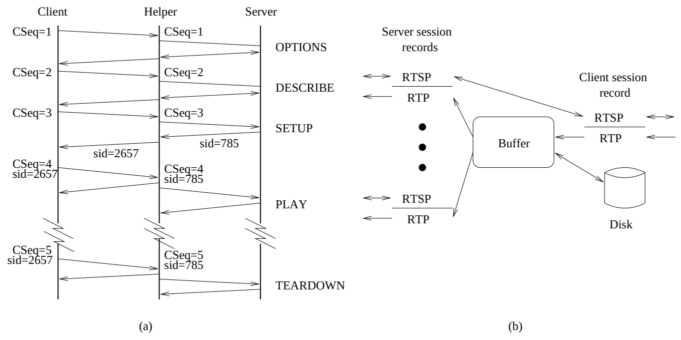


Figure 7. (a): RTSP message exchange across client, helper and server. (b): Session records at a helper for sessions with the clients and with the server, and the shared data buffer.

We now discuss the sequence of RTSP messages exchanged between a client, a helper, and a server, in cases where no data or only partial data for the media object is obtained from the local cache (Figure 7(a)). Note that the client-helper session is independent of the helper-server session: the two sessions have different RTSP session identifiers as well as different RTP SSRC (synchronizing source) identifiers. Translation of certain headers in the RTSP messages from the clients is thus necessary before forwarding them to the server. The helper forwards the OPTIONS, DESCRIBE, and SETUP message requests from the client to the server and forwards responses from the server to the client after suitable translation of CSeq (sequence number), Session, and Transport headers. The helper forwards the PLAY request to the server with a modified Range header based on how much data is available from the local buffer or disk. A TEARDOWN message is forwarded to the server once all the requested data is received from the server. If a local buffer has already been created to serve data in the requested range, then a TEARDOWN is sent to terminate the helper-server session for this request. In the meantime, data is streamed out of the local buffer.

Figure 7(b) illustrates the data structures used in the scenario where one local buffer is used to serve multiple client requests. In this case, a single session with the server is used to serve multiple client sessions thereby achieving request aggregation using the buffer.

The RTSP implementation in media players and servers from Real Networks [9] uses proprietary headers with encrypted header fields, which forced us to blindly forward OPTIONS, DESCRIBE, and SETUP messages to the server for every client request even if the data was available in the helper. While we would have preferred not to contact the server for URLs for which data is available in the helper, we are forced to do so in order to make our helper interoperable with Real players and Real servers. Note, however, that this additional delay incurred for every client request is negligible: client start-up latency is typically around 5-10 seconds, while the round-trip time for a RTSP message exchange is typically around 100 ms.

The data being streamed by the helper to a particular client might be obtained from different sources. For example, the initial segment of a media object may be cached at the helper while the remaining data must be obtained from the host server: in particular, RTP header fields (SSRC, sequence number, timestamp) will be chosen independently for the initial segment and the rest of the data. It is the responsibility of the helper to compose an outgoing data stream with meaningful RTP headers using data from different sources. Details of stream composition are discussed in Section 3.3.4.

3.3 Buffer management

The module for Buffer Management is the central multiplexing-demultiplexing point of the helper. This module serves as the data source for the RTSP/RTP server module and as the data sink for the RTSP/RTP client module. The Cache Management module interacts with this module when recording data to the cache and reading data from the cache. The Buffer Management module is currently implemented in the user level and we have not yet addresses issues related to improving the performance of the data path involving memory, disk, and network devices [3].

3.3.1 Buffer organization

The Buffer Management module manages a pool of buffers in main memory, where each buffer contains data for a media object within a certain time range. At any instant, there could be multiple buffers in the pool containing data for the same media object, each for a different time range. Also, a buffer could be the data source for servicing multiple clients. The source of data for the buffer itself could be the local disk, another helper or the server. This corresponds to the classical producer-consumer problem, where the size of the shared buffer is bounded. Here, the size of the buffer is specified in time units.

As each buffer can be filled with data from multiple sources, it is composed of multiple segments each associated with a data source and a time range for which data is fetched from the source into the segment. Each segment contains an array of RTP packets corresponding to the time range associated with the segment. As the first sequence number and time stamp of a RTP stream can be chosen randomly by the source, the helper needs to keep track of them to perform appropriate translation before forwarding them to different clients.

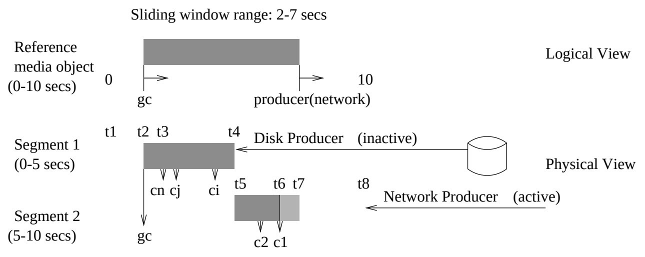


Figure 8. Buffer organization: logical and physical views.

Figure 8 shows an instantaneous snapshot of the memory buffer created to serve data for a media object in the range [0s, 10s]. The buffer has data in the playback range [2s, 7s]. The right edge of the sliding window moves as the network producer inserts more data into the buffer, while the left edge moves as the garbage collector (gc) frees data from the buffer. While this is the logical view, the figure also shows the actual implementation in terms of data structures: this buffer is composed of two time segments $[t_1, t_4]$ and $[t_5, t_8]$; data for the former segment is obtained from local disk and for the latter from the network. $t_1, t_2, ..., t_8$ represent RTP timestamps of data packets. The snapshot shows that the disk producer is currently inactive after having fetched the requested data from disk, while the network producer is actively filling the buffer with the remaining data. This buffer has n consumers: $c_1, c_2, .., c_n$. c_1 and c_n are currently consuming data packets with RTP timestamps t_6 and t_3 respectively. gc is the garbage collector event which frees RTP packets not needed by any of the consumers. The figure shows that gc has already freed RTP packets in the range $[t_1, t_2]$ from the buffer. The lightly shaded region in the figure between t_6 and t_7 represents the jitter absorption buffer.

3.3.2 Producer and consumer events

The rate at which data is streamed into and out of the buffer is governed mainly by the average bit rate of the media object. This is best implemented using timer events: events which insert data into the buffer are considered producer events and events that stream data out of the buffer are considered consumer events. These events could also be viewed as file and network events: file events are for transferring data from and to the local cached file, while network events are for transferring data from the server and to the clients.

As with the classical producer-consumer problem, buffer underflow and overflow problems need to be handled for each buffer. To address these issues, as shown in Figure 8, we maintain a list of consumers for each buffer sorted according to the position of the next packet to be streamed by the consumer. Buffer underflow and overflow can be handled for all consumers if it can be handled for the first and the last consumer. To handle underflow, we need to ensure that the producer event is ahead of the first consumer c_1 and to handle overflow, we need to ensure that the garbage collector gc is behind the last consumer c_n. The first and the last consumer of a buffer might keep changing with time because of range, rate, and arrival time heterogeneity across consumers using the buffer.

3.3.3 Garbage collection

The goal of the garbage collection algorithm is to free memory resources (RTP packets) allocated to a buffer when no longer needed by any of the consumers. As discussed earlier, each buffer is associated with a *buffer temporal distance* - Δ_b, which can also be seen as the size of the temporal sliding window. Δ_b for a buffer is statically chosen based on the configuration file when the buffer is created. Though Δ_b for a buffer is constant, the number of RTP packets within the temporal sliding window might vary as it slides along the timeline of the media object (for variable bit rate media, for example). Thus, the buffer cannot be implemented as a simple ring buffer.

One approach would be to associate a reference count with each RTP packet in the buffer which is equal to the number of consumers which are yet to "consume" this packet [10]. The reference count for a RTP packet is decremented every time it is forwarded by a consumer; the packet is freed once the reference count becomes zero. When the set of consumers associated with a buffer changes dynamically (consumers joining and leaving the set dynamically), this approach involves updating the reference count of relevant packets in the buffer each time a consumer joins or leaves the buffer.

We adopt an approach which attempts to retain minimal data for each buffer without the reference count overhead. Each buffer is associated with a garbage collection event (gc) which is first created when the buffer is filled with data corresponding to Δ_b, and is responsible for freeing the RTP packets after they have been forwarded by the last consumer (c_n) of the buffer.

The data insertion rate by the producer and the data freeing rate by the gc correspond to the streaming rate for the media object, which can also be seen as the sliding rate of the window. While the gc aims to free RTP packets at the media streaming rate, it cannot free the packets before they are consumed by the last consumer. In order to facilitate this, the consumer list is maintained in a sorted order according to the position of the data being consumed by each consumer: in Figure 8, c_1 is the first consumer and c_n is the last consumer. The buffer is freed once all the RTP packets in the buffer are freed by the gc.

3.3.4 Outgoing stream composition

Unlike a server, the helper might not have the entire media object in its local disk and might have to compose an outgoing stream by fetching the prefix data from the local disk and the rest of the data from the server. Therefore, special care is needed to create RTP headers for data streamed by the helper.

RTP packets obtained from a particular source begin with a random timestamp and a random sequence number chosen by the source, and have a unique SSRC (synchronizing source identifier) [12]. When data is streamed from a single buffer to multiple clients using multiple consumer events, each consumer uses a distinct RTP stream to transfer data. Each consumer c_i picks its own random timestamp and sequence number for the first outgoing RTP packet and a distinct SSRC for the outgoing stream. Thus, each consumer needs to calculate the timestamp and sequence number fields for every RTP packet forwarded from the buffer based on the values for the packet in the input stream and the values chosen by this consumer for the first packet.

Figure 9 shows an example where the outgoing RTP stream is composed of RTP streams from two different sources. Here, we consider a media clip which is 10 seconds long: the initial 3 seconds are cached at the helper's disk while the rest of the clip is obtained from a remote server. Assuming a constant bit rate, 1 RTP timestamp unit is equivalent to 1 millisecond and every RTP packet contains data equivalent to 16 milliseconds. 88221 and 91205 are the timestamps of the first and last RTP packets obtained from the disk; t is the timestamp of the first packet not cached on disk, shown as a shaded box. 36057 and 43041 are the timestamps of the first and last RTP packets obtained from the network. We also assume that the data in packet with timestamp t is the same as the data in packet with timestamp 36057.

The SSRC field in the outgoing RTP headers can be easily set to the unique value chosen for the stream. However, the sequence number and timestamp fields need to be set carefully: the sequence number and the timestamp for the first packet of the outgoing stream can be chosen randomly. They are incremented for each subsequent packet following the same pattern as in the source segment.

A special case occurs when a consumer switches from a segment with data from disk to a segment with data from

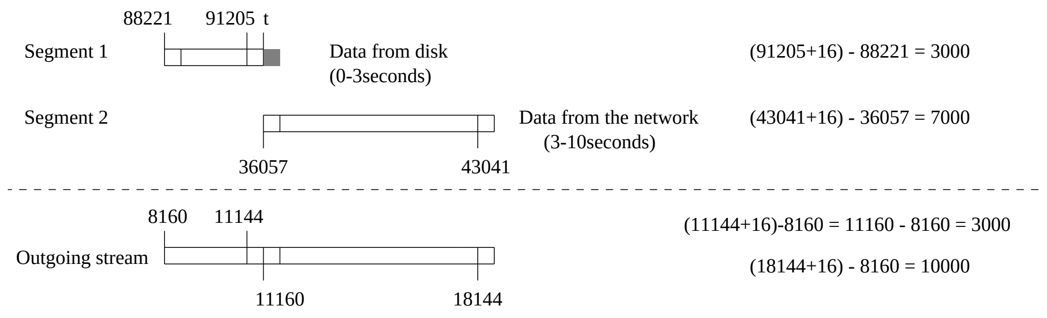


Figure 9. RTP timestamp translation for a stream composed of two contiguous segments.

the network. Before forwarding the first packet obtained from the network, the helper needs to determine the outgoing sequence number and timestamp for this packet. This translation can be successful if the beginning of the data from the network corresponds to the end of the data from the disk. There should not be any gap nor any overlap. As shown in Figure 9, the outgoing timestamp of the last packet from disk is $11144 = (8160 + (91205 - 88221))$ and the outgoing timestamp of the first packet from the network is $11160 = (11144 + 16)$ since the timestamp increases by 16 for every RTP packet.

3.4 Cache management

The functionality of the Cache Management module includes: mapping a URL to a filename, recording; reading data within a time segment of a media object; and deleting files which have not been used in the recent past. As discussed in Section 2.1, a specific time segment of a media object is chosen to be cached on local disk based on the *prefix caching* technique. A cached copy of a media object is replaced from the cache based on a cache replacement policy (for example, LRU). Cache management also ensures that concurrent read and record requests for the same media object are handled appropriately.

To make the cache fault-tolerant, an index file containing meta-information about the cached media objects and their corresponding local cache file names is maintained. Hashing techniques used in the organization of the index file also aids faster lookups for media objects. The content of this index file is flushed onto the disk frequently. When a helper restarts (after a power failure, for example), the index file is used to remove any cache files with inconsistent data.

Our goal is to make the helper independent of the media encoding mechanism. Hence, we cache the data in terms of RTP packets; we store the RTP payload and minimal header information like payload type, sequence number, timestamp, marker bit, and the RTP header extension corresponding to every packet cached. With this minimal information, new RTP packets for the outgoing stream with appropriate RTP header and header extensions can be created. In this context, the cached files have a generic format conducive to network transfer and can be considered to be in “RTP file format”, independent of the media encoding mechanism used.

As media objects are very large, it is not always possible to cache the entire object on local disk. In addition, requests can start from any instant in a media object, resulting from VCR operations for example. Therefore, the format of a cached file is designed to support the recording/reading of multiple, non-overlapping time segments of the corresponding media object.

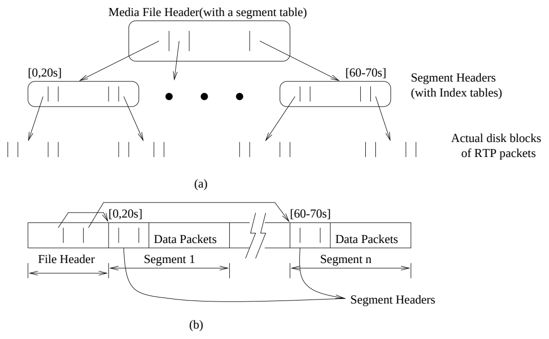


Figure 10. (a): Logical structure of a cached file. (b): Physical layout of a cached file.

A cached file is structured as follows:

- *Media file header:* Each cached file begins with a header which contains information specific to the media object stored in the file. Typically, this is obtained from the server in response to the RTSP DESCRIBE request. In addition, the header also maintains a segment table with file offsets marking the start of different time segments in the file.
- *Non-overlapping time segments:* Following the file

header are non-overlapping time segments, each of which has a segment header containing segment specific information: start and end timestamps for the segment; start sequence number; and number of RTP packets, among others. Data packets belonging to a segment are organized in terms of consecutive blocks. The segment header also maintains an index table with pointers to individual blocks of packets facilitating faster access to data within a subrange of a segment.

Figure 10(a) illustrates the logical structure of a cached file viewed as a tree. The media file header with the segment table is the root of this tree and headers of various time segments with corresponding index tables are its children. Figure 10(b) shows the actual layout of the file.

4 Performance

We have implemented the helper on FreeBSD and each helper is currently a single process. The current implementation has been tested for interleaved RTP data transfer along with RTSP control messages on a single TCP connection.

The performance of the system from both the network and user perspectives is evaluated in this section. We focus on the effectiveness of employing request aggregation to handle the heterogeneity problem as outlined in Section 2.3. In particular, we look into the arrival time and media object heterogeneity issues handled by varying the buffer temporal distance (Δ_b) and the size of prefix cache for each movie.

We use a client-helper-server setup, where the client simulates multiple media players by generating requests for media objects according to a uniform distribution model over time. We use the Real server [9] and a media client which disregards the incoming data; the media objects are encoded in MPEG requiring us to use the plug-in from DBC Bitcasting [4] with the Real server. We run the Real server on a Sun Ultra-4 workstation with 4 processors, 1GB main memory and Sun OS version 5.6. We run the client on a 300MHz Pentium Pro with 250MB of main memory and the helper on a 400MHz Pentium II with 250MB of main memory, both running FreeBSD. The helper and the client are on a 10Mbps Ethernet interconnected by a router to another 10Mbps Ethernet with the server. We use 12 media clips encoded in MPEG, whose lengths vary from 40 to 70 seconds.

4.1 Network load and server load

By virtue of request aggregation achieved by the helper, both server load and network load between the helper and the server (henceforth referred to as network load) should be decreased. The *traffic reduction ratio* is the indicator of request aggregation achieved by the helper in a given time interval. It is defined as the ratio of reduced network load with the helper to network load without the helper. Thus,

$$R = (D_{out} - D_{in})/D_{out} \tag{3}$$

where D_{out} is the total amount of data transferred from the helper to the client, and D_{in} from the server to the helper. A larger value of R indicates larger server load and network load savings. For example, $R = 0.4$ represents a 40% reduction in network and server loads offered by this helper.

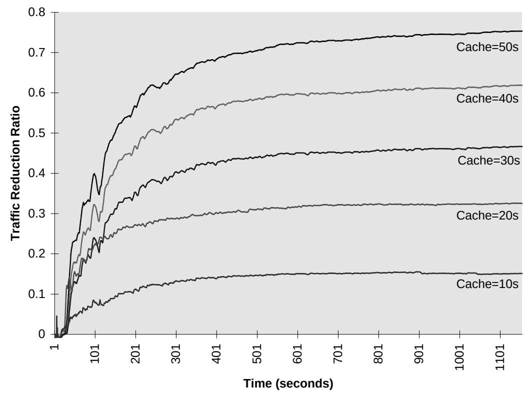


Figure 11. Prefix caching benefits: reduction in network and server loads with increasing prefix cache size.

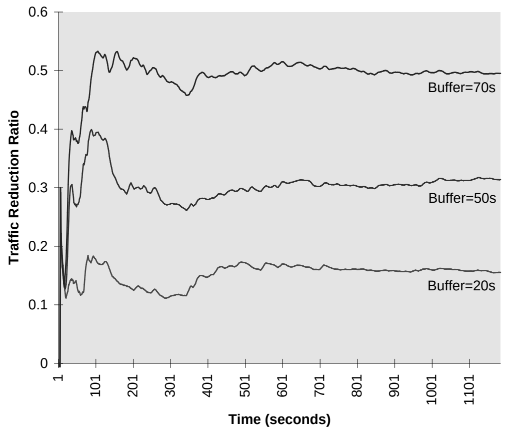


Figure 12. Buffer request aggregation benefits: reduction in network and server loads with increasing ring buffer size.

Figure 11shows the cumulative traffic reduction ratio plotted against time for varying sizes of prefixes cached on local disk. Requests are generated uniformly by the client with an average inter-arrival time of 15s and the URLs are selected from a set of 12 MPEG media objects according to the Zipf distribution. No cache replacement policy is enabled for this experiment, which essentially simulates infinite disk space. In order to preclude any request aggregation offered by memory buffers, we set Δ_b to 10s which is less than request inter-arrival time. As expected, the traffic reduction ratio increases with increasing prefix cache size, thereby decreasing the server and network loads. We notice that a load reduction of up to 75% is achieved with a prefix of size $50s$ for the chosen sample of media objects.

Figure 12 shows the cumulative traffic reduction ratio against time while Δ_b for individual buffers is varied; requests are generated uniformly by the client with an average inter-arrival time of 10s. Disk caching is disabled to preclude any request aggregation by data from disk. Here again, we notice that the ratio increases with increasing Δ_b, thereby decreasing the network and server loads. We notice that a load reduction of up to 50% is achieved with a Δ_b of $70s$ for the chosen sample of media objects.

Request aggregation in the buffer and prefix caching are complementary mechanisms and contribute to the reduction of server and network loads. Increased benefits from buffer aggregation can be observed when requests for the same media object arrive closely in time relative to Δ_b, while benefits from the cache can be increased by increasing the prefix size.

4.2 Client start-up latency

To see the effectiveness of the data transfer rate control mechanism applied at the helper, we measure client start-up latency by varying K_1 and the ratio of streaming rates b/r as discussed in Section 2.3. The measured start-up latency is the time taken to fill the client's playout buffer of length $K = 5s$.

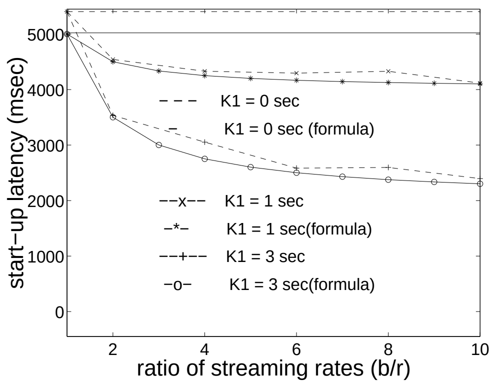


Figure 13. Improvement on startup latency

The solid lines in Figure 13 represent plots from formula (2) in Section 2.3 and the dotted lines represent our measurements. The parameter setting in the formula is based on measurements: $d_1 = 10ms$ and $d_2 = 1ms$. Measurements show the improvement on startup latency with increasing K_1 and b/r; a reduction in startup latency of up to 50% is achieved by streaming the initial $3s$ of the media object at 10 times its streaming rate. Also note that the plots from the measurements match closely with the plots for the formula.

5 Related work

We aim to improve service quality of streaming media as perceived by end users by introducing helpers in the network. Each helper employs prefix caching, request aggregation, and rate control mechanisms to improve start-up latency at end users, while also reducing the network and server loads.

[14] presents a prefix caching scheme similar to our approach. In contrast to the workahead smoothing technique, we increase the streaming rate for the prefix of a clip to improve the start-up latency even further. [11] describes a proxy caching mechanism for layered-encoded multimedia streams in the Internet. The proxy attempts to replay a quality-variable cached stream while performing quality adaptation dynamically. The above mechanisms are complementary to our approach and can be easily accommodated in our system. [17] describes a video staging technique which is useful in maintaining a constant bit rate stream between the proxy and the server. This scheme, however, relies on the server being aware of what data is available in the proxy cache and hence it is not transparent to the server.

Service Aggregation [16] aims at aggregating users into a single channel by using rate adaptation and content insertion techniques along with IP multicast. While our sheme utilizes the disk and memory resources at the helper for request aggregation, this scheme requires the end user to have a larger memory. Higher levels of aggregation can be achieved by combining our scheme with some of the above schemes and their suitability at the helper is currently under investigation.

Extensive work in the context of multimedia servers is quite relevant to the helper design [6, 5, 3, 2]. In addition to efficient management of the local memory and disk resources, a helper is also concerned with the reduction of network and server loads. Chaining [15] and Patching [8] employ proxy cooperative schemes, where proxies share their local caches thereby further reducing the server load. Our helper design is aimed for a distributed caching architecture reported in [7].

6 Conclusions

This paper discusses the design and implementation of a helper for caching streaming media over the Internet. The helper strives to improve end users' perceived quality of multimedia streams by prefix caching, request aggregation, and rate control. By virtue of its position in the path between the clients and the servers, the helper attains high levels of request aggregation by using its memory and disk resources to overcome heterogeneity in client requests, thereby reducing the server and network loads. The proximity of the helper to the client, along with increased streaming rate of the prefix of the media object improves the start-up latency at the client.

Performance measurements clearly show the benefits of prefix caching, buffer aggregation, and rate control mechanisms. For the sample set of media objects of about 1 minute in length, the system demonstrates up to 75% reduction in network and server loads by caching prefixes of length $50s$ compared to a helperless system. A 50% reduction in network and server loads is demonstrated by using request aggregation with buffer of $70s$. By streaming the initial $3s$ of a media object at 10 times the streaming rate, a 50% improvement in the start-up latency is achieved. The implementation of the helper incorporates most of the issues discussed in this paper and is inter-operable with the popular Real players and servers. As observed in [1], user access patterns, media characteristics (length of the clip and the bit rate, for example), among others play a key role in the benefits achieved by any system aimed at improving streaming media service quality. In this context, we are currently working towards productizing the helper and deployment in the Internet.

Acknowledgments

We would like to thank Jonathan Lennox, Jonathan Rosenberg, and Henning Schulzrinne for helpful discussions on RTSP and RTP. We would also like to thank Milind Buddhikot, Ray Miller and Mike Vernick for their valuable suggestions to improve the paper.

References

[1] S. Acharya and B.Smith. An experiment to characterize videos on the world wide web. In *Proc. ACM Multimedia*, Sept 1998.

[2] K. Almeroth and M. Ammar. On the use of multicast delivery to provide a scalable and interactive video-on-demand service. *Journal on Selected Areas of Communication*, Aug 1996.

[3] M.M. Buddhikot, X.J. Chen, D. Wu, and G. Parulkar. Enhancements to 4.4 bsd unix for networked multimedia in project mars. In *Proc. IEEE Multimedia Systems*, Jun. 1998.

[4] Digital Bitcasting Corp. Internet homepage. http://www.bicasting.com, 1999.

[5] A. Dan and D. Sitaram. A generalized interval caching policy for mixed interactive and long video environments. In *Proceedings of IS&T SPIE Multimedia Computing and Networking Conference, San Jose, CA*, January 1996.

[6] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *Multimedia Systems*, 4(3):51–58, June 1996.

[7] M. Hofmann, E. Ng, K. Guo, S. Paul, and H. Zhang. Caching techniques for streaming multimedia over the internet. Technical Report BL011345-990409-04TM, Bell Laboratories, April 1999.

[8] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proceedings of ACM Multimedia '98*, Bristol, England, September 1998.

[9] Real Networks. Internet homepage. http://www.real.com, 1999.

[10] U. Ramachandran, R.S. Nikhil, N. Harel, J.M. Rehg, and K. Knobe. Space-time memory: A parallel programming abstraction for interactive multimedia applications. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[11] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet. submitted to IEEE Infocom, 2000.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. Internet Request for Comments 1889, January 1996.

[13] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). Internet Request for Comments 2326, April 1998.

[14] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of IEEE Infocom'99*, New York, USA., 1999.

[15] S. Sheu, K. A. Hua, and W. Tavanapong. Chaining: A generalized batching technique for video-on-demand systems. In *Procedings of IEEE International Conference on Multimedia Computing and Systems*, Ottawa, Ontario, Canada., 1997.

[16] D. Venkatesh and T.D.C. Little. Dynamic service aggregation for efficient use of resources in interactive video delivery. In *Proc. of the 5th NOSSDAV*, Nov 1995.

[17] Y. Wang, Z.-L. Zhang, D. Du, and D. Su. A network conscious approach to end-to-end video delivery over wide area networks using proxy servers. In *Proceedings of IEEE Infocom*, April 1998.