Mocha: A Quality Adaptive Multimedia Proxy Cache for Internet Streaming

Reza Rejaie
AT&T Labs - Research
Menlo Park, CA. 94025
reza@research.att.com

Jussi Kangasharju
Institut Eurecom
Sophia Antipolis, France
kangasha@eurecom.fr

ABSTRACT

Multimedia proxy caching is a client-oriented solution for large-scale delivery of high quality streams over heterogeneous networks such as the Internet. Existing solutions for multimedia proxy caching are unable to adjust quality of cached streams. Thus these solutions either can not maximize delivered quality or exhibit poor caching efficiency.
This paper presents the design and implementation of *Mocha*, a quality adaptive multimedia proxy cache for layered encoded streams. The main contribution of Mocha is its ability to adjust quality of cached streams based on their popularity and on the available bandwidth between proxy and interested clients. Thus Mocha can significantly improve caching efficiency without compromising delivered quality. To perform quality adaptive caching, Mocha implements *fine-grained replacement* and *fine-grained prefetching* mechanisms. We describe our prototype implementation of Mocha on top of Squid and address various design challenges such as managing partially cached streams. Finally, we validate our implementation and present some of our preliminary results.

1. INTRODUCTION

Most of today's Internet streaming applications have a client-server architecture where a server pipelines a requested stream to a client through the network. The client-server architecture for Internet streaming has two major limitations. First, it does not scale to a large number of clients because streaming applications consume network bandwidth along the path from the server to the client for the entire session. Second, the quality of pipelined stream is limited to the bottleneck bandwidth along the server-client path.

To achieve scalability and deliver high quality streams, multimedia content should be maintained close to interested clients. Proxy caching of multimedia streams is a client-oriented solution that addresses both limitations simultaneously. Similar to Web proxy caching, caching of popular streams at a proxy substantially reduces the load on the network (and the server), which in turn accommodates scalability. Furthermore, since a proxy is located close to its clients, it can effectively avoid network bottleneck and maximize delivered quality and accommodate client bandwidth heterogeneity. The ability to adjust the quality of a cached stream is a crucial requirement for multimedia proxy caching mechanisms in heterogeneous networks such as the Internet that has not been well-understood. To justify this claim, consider a proxy that has three clients with different bandwidth (Figure 1). To maximize delivered quality of a cached stream s to all clients, the proxy should be able to provide appropriate versions of stream s that can be pipelined to each client. This implies that caching mechanisms should be *quality adaptive*. There are two alternatives to achieve quality adaptive caching: 1) Caching different encoded versions of stream s with different quality, or 2) Caching layered encoded version of stream s where the appropriate quality (*i.e.*, number of layers) for each client is determined by its bandwidth[1]. Layered encoding is an efficient and flexible way to adjust quality of cached streams without losing cache performance due to the following reasons: 1) Multimedia streams are orders of magnitude larger than typical Web objects. Thus caching multiple versions of each stream could significantly reduce cache utilization; 2) Layered organization provides an opportunity to improve delivered quality of a cached stream on-the-fly where lower layers are played back from the cache and higher layers are delivered from the server.

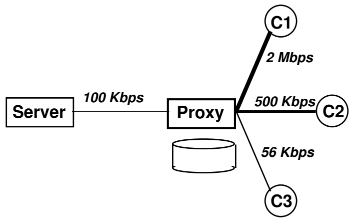

Figure 1: Multimedia Proxy Caching

Therefore the design of a multimedia proxy caching mechanism should address two key issues:

1. *Which streams are sufficiently popular to be cached?*
2. *What is the appropriate quality for each cached stream?*

The first issue in essence is a Web cache replacement problem. The second problem, however, addresses the notion of *quality* for cached streams as a new dimension in design of caching mechanism that does not exist in Web caching schemes. If a majority of clients can only afford to receive a low quality version of stream s, the proxy can cache only the low quality (and thus smaller) version of the stream to improve cache utilization. To adaptively increase or decrease quality of cached streams, the proxy should implement *fine-grained prefetching* and *fine-grained replacement*, respectively. Furthermore the server should provide access to a portion of a layered encoded stream.

Most of the work on design and development of multimedia proxy caches has only focused on the first issue and treated multimedia streams in an atomic fashion similar to Web objects. These approaches could result in poor cache utilization since multimedia streams are orders of magnitude larger than typical Web objects. Admittedly, this is in part due to lack of support for layered encoded streams by major content providers. Most of the previous work on multimedia proxy caching address neither network bandwidth heterogeneity nor the need for rate and quality adaptation. Therefore they are not suitable for the Internet. To the best of our knowledge, Mocha is the first quality adaptive multimedia proxy cache. Design and evaluation of multimedia proxy caches are still immature in compare to Web caching schemes. Judging based on the work on Web caching schemes, design and evaluation of multimedia proxy caching mechanisms clearly require substantially more investigation.

This paper presents the design and implementation of *Mocha*, a multimedia proxy cache for layered encoded streams on top of Squid[2]. Mocha's key contribution is its ability to perform quality adaptive caching. Mocha caches popular streams and adaptively adjusts quality of cached streams based on both stream popularity and available bandwidth to interested clients. Mocha leverages layered structure of streams to implement fine-grained replacement and fine-grained prefetching mechanisms. Therefore, Mocha is able to maximize delivered quality for a group of heterogeneous clients without compromising cache space utilization. We address the high level architecture of Mocha and discuss the main issues and challenges in the design of key components of the architecture. We validate our prototype through various experiments, and present some of our preliminary results.

Mocha requires a client-server architecture that supports layer-encoded streaming in order to adapt quality of cached streams. Mocha can also manage non-layered encoded streams. However, it can not adjust quality of cached streams in this scenario. We have prototyped a modular client-server architecture [3] for delivery of layered encoded stream as shown in Figure 2. Client and server use RTSP [4] for signaling. The server performs congestion control [5] to determine fair share of bandwidth. Then a layered quality adaptation mechanism [6] matches the number of transmitted layers with average bandwidth. Although we are interested in unicast streaming, we used RTP [7] for data packets and RTCP for ack packets. Each layer is transmitted through a separate RTP session. However, congestion control is collectively performed across all RTP sessions. The client receives packets of different layers and rebuilds the stream in a reorganization buffer before sending the stream to the display.

The rest of this paper is organized as follows: Section 2 justifies why we developed Mocha on top of Squid and sketches the high level architecture of Mocha by describing its request management. Section 3 addresses design issues

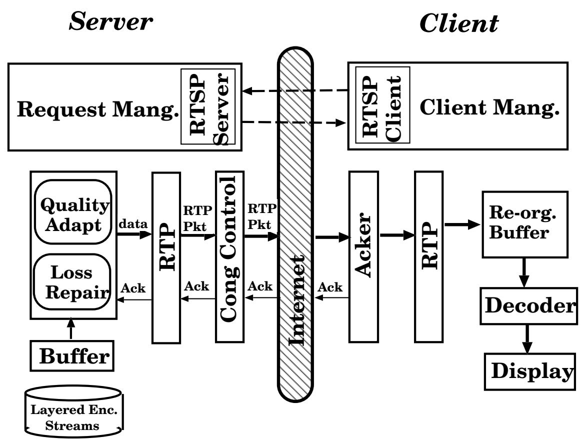

Figure 2: Client-server Architecture

and challenges for the key components of Mocha. Section 5 briefly reviews related work. In Section ??, we report some preliminary experimental results. Finally, Section 6 concludes the paper and addresses our future directions.

2. ARCHITECTURE

Mocha was developed on top of open-source code Squid[2]. Our main motivation was to leverage generic components of Squid (such as request processing, storage and memory management, and general utility routines for memory allocation and event handling) that can be reused for multimedia caching with little or no modification. For example, we took advantage of similarity in message processing between HTTP and RTSP, and simply replace HTTP routines with their RTSP correspondent. Therefore we only needed to add those components that are required to cache and replay layered encoded streams. This substantially reduced our prototyping and debugging phases. Obviously, building on top of Squid introduced a few restrictions and limitations. For example, we needed to extend storage management routines such that all layers of a cached stream are collectively viewed as a single object by Squid. More importantly, Squid's file system is probably not optimized to store multimedia objects. Thus it is likely that our current implementation does not scale to a large number of clients. Fortunately, Squid has a modular structure and we can replace its file system routines whenever it becomes a bottleneck. Another key requirement for efficient handling of streaming objects is management of disk bandwidth. We need to add such a management mechanism to achieve high performance. Placement and retrieval of multimedia streams have been extensively studied in the context of multimedia servers (*e.g.*, [8]). Our goal is to study transport issues for object management across the Internet rather than well-understood local resource management issues at the proxy.

Mocha appears as a client for a server and as a server for a client. Figure 3 depicts the internal architecture of Mocha as a combination of a client and a server. We explain the functionality of individual components by describing request management in Mocha.

2.1 Request Management

Request manager(RM) module handles all the RTSP sig-

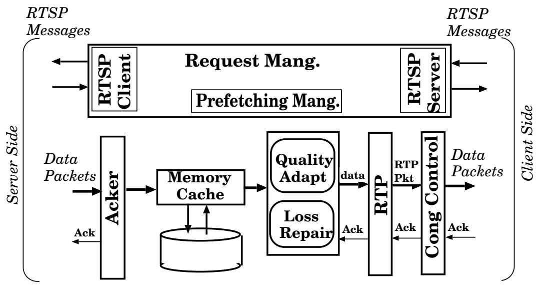

Figure 3: Internal Architecture of Mocha

naling between Mocha and client or server. Upon arrival of a RTSP/SETUP request for a stream s, RM checks the availability of s in the cache, and one of the following scenarios occurs:

- **Cache Miss:** If s is missing from the cache, RM relays all RTSP message in both directions between client and original server (or another proxy depending on configuration). Mocha also relays data and acknowledgment(ACK) packets in both directions as shown in Figure 4 Obviously, packet relaying process will introduce a delay but we expect the delay to be small under moderate load on the proxy. Therefore, the server effectively measures losses and round-trip-time(RTT) of the server-client connection, and adapts its transmission rate and delivered quality (*i.e.*, number of layers) accordingly. This implies that on a cache miss, the session is end-to-end and quality of the played back stream is limited by the bottleneck bandwidth, *i.e.*, proxy can not improve delivered quality on a miss. Notice that the original server can send either a stored or even a live stream. Mocha can intercept all transmitted data packets and cache a copy of the delivered stream.

- **Cache Hit:** If a copy of s is available in the cache, Mocha acts as a server and directly replies to all RTSP

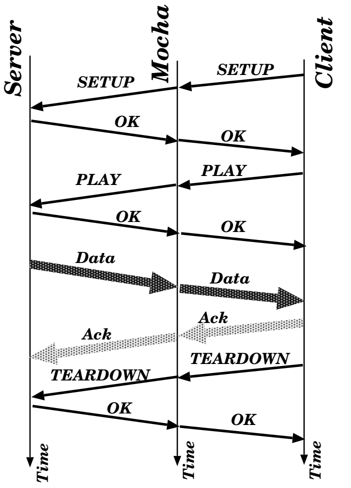

Figure 4: RTSP signaling and data delivery on a miss

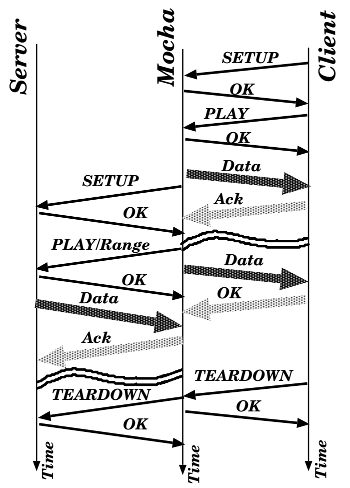

Figure 5: RTSP signaling and data delivery on a hit

messages as shown in Figure 5. Upon arrival of PLAY request from client, Mocha initiates delivery of cached stream while performing rate and quality adaptation based on the state of the proxy-client connection. If the quality of the cached stream is lower than the maximum deliverable quality to the client, prefetching manager establishes a prefetching session to the server to prefetch missing parts of an existing layer as well as higher layers of requested stream on-the-fly. Thus on a cache hit, Mocha should manage both playback and prefetching sessions such that prefetched segments arrive before their play out times to be properly merged with playback stream. Fine-grained prefetching is discussed in more details in subsection 3.2.

3. MAIN COMPONENTS

In this section, we discuss the design and implementation of three key components that are unique in Mocha: *Object Management*, *Fine-grained Prefetching*, and *Fine-grained Replacement.*

3.1 Object Management

Mocha caches RTP packets instead of their raw payload. Although caching header of RTP packets reduces cache space utilization, the proxy does not need to deal with various payload formats and becomes content-independent. Mocha relies on RTP sequence number to detect missing segments of a layer, and uses RTP time-stamp (and marker bit) to store and play back RTP packets properly. Mocha can also maintain a few well-accepted RTP profiles to properly interpret RTP header information.

One of the main challenges in the design of storage management for Mocha was to store and access partially cached layers of a single stream efficiently. Since Web caches store or flush an object in an atomic fashion, we needed to extend Squid's data structures to maintain information about cached segments of all layers. Furthermore, all layers of each stream should be collectively viewed as a single object by Squid in order to reuse its basic object management features (*e.g.*, checking hit or miss scenarios). Squid maintains a StoreEntry for each object where object-specific informa-

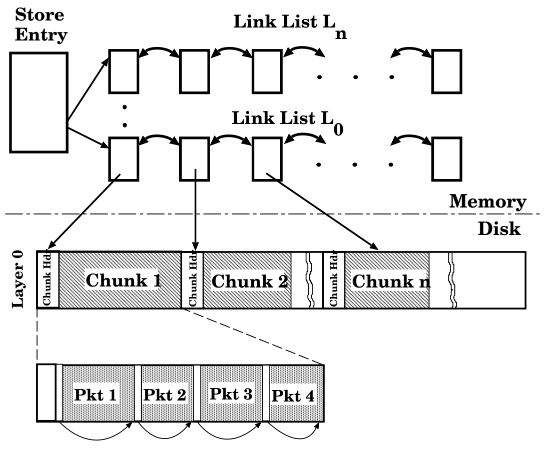

Figure 6: Data structures for object management in Mocha

tion is kept. Figure 6 shows how we extended Squid's data structures to manage layered encoded streams in Mocha. Packets of each layer of a cached stream are stored in a separate file. Each file contains a collection of *chunks* where a chunk consists of a group of contiguous packets. Location of a packet in a chunk can be easily calculated if all packets have the same size. But in a general case, packets can be variable in size. To allow quick traversal within a chunk, we interleaved pointers to the next packet between every two packets in each chunk. In order to quickly locate a specific chunk, Mocha maintains a linked list in memory for each layer of an active stream that is being played back. Each element of the linked list points to one or a group of chunks on disk. Thus by traversing the linked list, Mocha can rapidly identify the location of a specific chunk on disk and minimize disk access. The bigger the chunk size, the shorter the linked list, but it takes longer to reach a specific packet within a chunk.

Mocha treats a chunk as a an atomic unit, *i.e.*, all packets of a chunk are cached or replaced together. When a hole in sequence number is detected or N contiguous packets arrive, all the previously received packets are cached as a chunk. While chunks can have variable sizes, Mocha limits the maximum number of packets in a chunk (*i.e.*, N). When a missing packet arrives during prefetching, two small adjacent chunks can be consolidated. The interactions between consolidation and fine-grained replacement result in a set of pseudo-balanced chunks.

3.2 Fine-grained Prefetching

Mocha implements online fine-grained prefetching in order to improve the delivered quality of a cached stream. When the quality of a cached stream is lower than the maximum deliverable quality to an interested client, Mocha initiates a connection to the server and acts as a client. Then it sends prefetching requests for missing pieces of active layers that are likely to be needed during the playback. Each missing packet should be prefetched before its play out time. Since the prefetching session is congestion controlled, the available server-proxy bandwidth is not known a priori and could vary in time. The available prefetching bandwidth should be used efficiently to deliver missing packets in a prioritized fashion

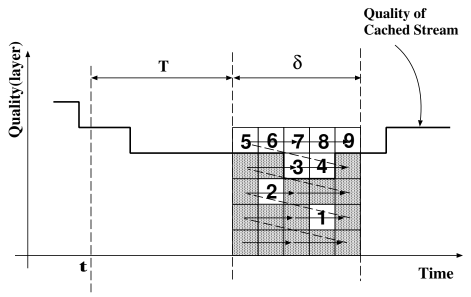

Figure 7: Online Prefetching in Mocha

such that prefetching session remains loosely synchronized with the playback session.

To achieve this, we devised a sliding window approach to prefetching that is illustrated in Figure 7. At time t during playback, the prefetching manager examines a window of time in the future ($[t + T,\ t + T + \delta]$) to identify required packets that are missing. If required segments are in the cache, they are fetched into the memory cache to avoid any potential delay during disk access. At the same time, Mocha sends a single prefetching request which contains an ordered list of all required but missing packets of this window. Requested packets are ordered based on their importance, *i.e.*, based on layer number and within a layer based on their play out time in a round-robin fashion as numbered in Figure 7. The server delivers requested segments in the specified order through a congestion-controlled connection. Thus, in the absence of sufficient prefetching bandwidth, only the most important segments are delivered. After p seconds, Mocha examines the next prefetching window and sends another prefetching request to the server. To keep playback and prefetching session loosely synchronized, each prefetching request will pre-empt any previous prefetching request. In summary, online prefetching has three parameters, 1) δ, Length of prefetching window, 2) T, look-ahead distance and 3) p, sliding period where $p \leq \delta$. If $p < \delta$, there is an overlap between adjacent windows which results in a more conservative use of prefetching bandwidth.

The "Range" header field of RTSP PLAY method was used to prefetch a group of contiguous packets of a layer. We extended the "Range" header field to carry multiple ranges of several layers in a single RTSP message. Another issue was sequential processing of RTSP PLAY requests. To allow a new prefetching request to pre-empt previous prefetching requests, we introduced a new type of Range field in RTSP that can be over-written by new Range requests.

Besides the data structures described in Section 3.1, Mocha maintains status of cached packets of all layers of a stream in a bitmap. The bitmap is used to implement the sliding window approach efficiently because missing packets can be identified without accessing the hard disk. At any point of time, only bitmaps of active streams are kept in the memory. We plan to add an off-line prefetching mechanism to Mocha.

3.2.1 Memory Caching

Squid has a built-in memory cache on top of storage that holds popular objects in a LRU fashion to minimize disk access. Mocha leveraged this memory cache with some modifications to improve cache performance. On a cache hit, the memory cache temporarily locks cached packets that are fetched from disk ahead of time as well as prefetched packets from the server until they are played out or their playout times expire. Therefore at any point of time a window of fetched or prefetched packets are maintained in the memory cache.

3.3 Replacement Policy

We modified Squid replacement policy to implement fine-grained replacement mechanism. Squid periodically invokes replacement routines and if the amount of total cached data is higher than a configured high-water mark, a sufficient number of least popular objects are evicted. We basically replaced the victim selection mechanism.

In order to implement fine-grained replacement, we need to define fine-grained popularity, *i.e.*, assign popularity values to pieces of a stream. Since quality of a cached stream is determined by number of cached layers, Mocha assigns popularity to individual layers. Popularity of layer i of stream s is defined as follows:

$$P_i(t) = \sum_{x\in[t-\Delta,t]}^{t} whit(x,i),$$

$$whit(x,i) = \frac{PlayTime(x,i)}{StreamLength(s)}$$

where $whit(i,x)$ is *weighted hit* of layer i during session x. We used the cumulative value of $whit$ across all sessions over a recent window of time ($[t-\Delta..t]$) as popularity of a layer. This definition of popularity captures both level of interest among clients and available bandwidth to interested clients[1]. Consequently lower layers of an unpopular stream might be flushed before higher layers of streams that are popular among high bandwidth clients. Furthermore, this definition guarantees that within a single stream, popularity monotonically decreases with layer number, *i.e.*, the victim layer is always the highest layer of a cached stream. Popularity of all cached layers are maintained in a sorted linked list, called *popularity list*. Thus a victim layer is always at the end of the list. At the end of each session, $whit$ and popularity of active layers are updated and their location in the popularity list is changed accordingly.

Although Mocha keeps track of popularity of individual layers, replacement is performed at a per chunk basis to achieve high cache space utilization. When the amount of cached data exceeds the high water mark, Squid invokes Mocha's replacement routine. The least popular layer is selected as victim and chunks of a victim layer are flushed from the end to the beginning, until total amount of cached data is less than the high water mark. The interactions between fine-grained replacement and sliding window approach to prefetching smooth out quality of cached streams.

fine-grained replacement can potentially result in thrashing where packets of a cached layer is flushed in order to make room for prefetched packets of a higher layer of the same stream. To prevent such behavior, Mocha employs a simple locking mechanism. All active layers that are being played back are locked for the entire duration of the session.

4. EXPERIMENTS

We are currently validating our prototype implementation. In this section, we briefly present some of our preliminary results to illustrate the basic features of Mocha. Our data set consists of 50 streams with 6 layers, and the bandwidth of all layers is 6 Kbps. Stream lengths were chosen randomly within the range of [30sec.. 180sec]. We generate a request sequence with 5000 requests and Zipf-like popularity distribution with proper temporal locality. Popularity of streams monotonically decreases with stream ID, *i.e.*, s_0 is the most popular. The cache size is 30% of the size of data set. The value of popularity window is infinite ($\Delta = \infty$). We use the topology in Figure 1, and a client-server architecture that is similar to Figure 2.

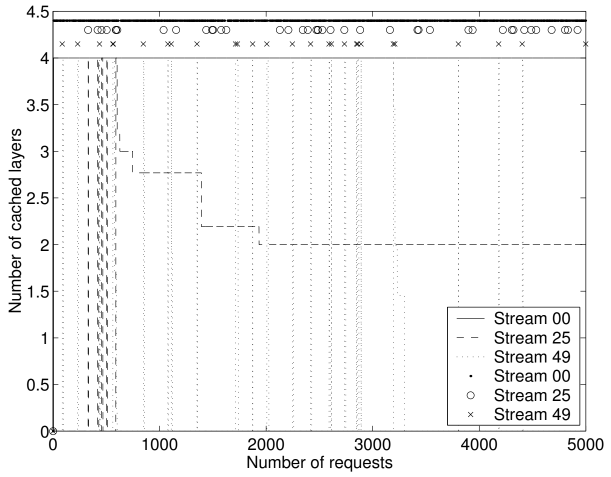

Figure 8: Quality of cached streams

First, we examine the behavior of fine-grained replacement mechanism when the online prefetching mechanism is turned off. We conduct an experiment with a single client where proxy-client bandwidth is only 24 Kbps (*i.e.*, 4 layers). Figure 8 depicts variations in quality of three cached streams (s_0, s_{25}, s_{49}) with minimum, moderate and maximum popularity during the experiment, respectively. The time of each request for these three representative streams is also shown at the top of Figure 8. This figure clearly illustrates the impact of stream popularity on dynamics of cache replacement. s_0 quickly becomes popular and all 4 layers are cached for the entire experiment. s_{49} never becomes sufficiently popular to stay in the cache, thus all layers of s_{49} are always played back from the server and are removed from the cache after a short period. After several close requests for s_{25} early in the experiment ($request\_number < 500$), all 4 layers of s_{25} are cached. Then its quality is gradually degraded since other streams become more popular than s_{25}. Since the fine-grained prefetching mechanism is turned off, higher layers of s_{49} are not prefetched even when it becomes more popular later in the experiment.

Next we turned on the fine-grained prefetching mechanism to examine its interactions with the fine-grained replacement mechanism in the presence of two heterogeneous clients with 36Kbps and 12Kbps bandwidth. We use a smaller data set with 20 streams where stream lengths are chosen randomly within the range of [30sec .. 180sec]. We generate a request sequence with Zipf-like popularity distribution and proper temporal locality. To examine the effect of client bandwidth, 70% of total requests for each stream are issued by the high bandwidth client and the rest are issued by the low bandwidth client. Cache size is 30% of the size of data set. Fig-

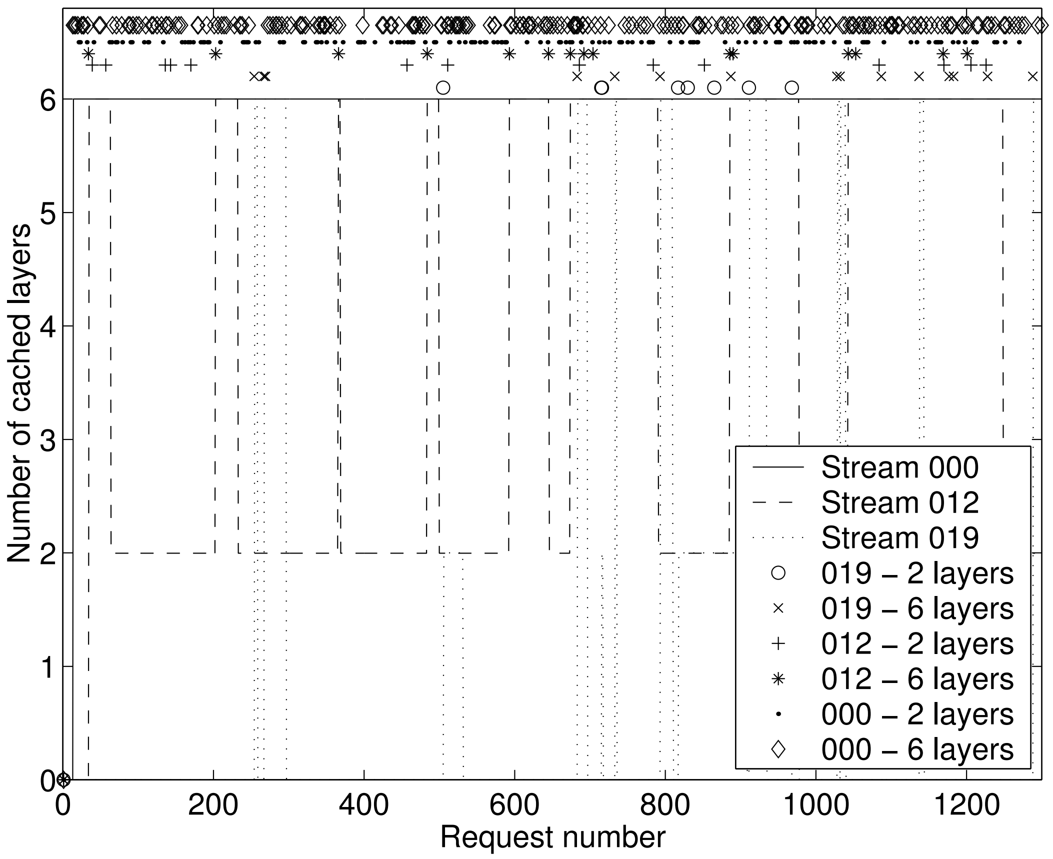

Figure 9: Quality of cached streams under prefetching and heterogenous bandwidth

ure 9 depicts the variations in quality of three representative streams (s_0, s_{12}, s_{19}) with minimum, moderate and maximum popularity during the experiment, respectively. The time of each high and low bandwidth request for these three representative streams are shown at the top of this figure. All 6 layers of the most popular stream (s_0) are brought into the cache at the beginning and stay in the cache for the rest of the experiment. All 6 layers of the moderately popular stream s_{12} are cached after the first request for s_{12} from the high bandwidth client. The following 2 requests for s_{12} from the low bandwidth client were served from the cache. Then the top 4 layers are replaced since they have not been requested sufficiently. However, the lower 2 layers of s_{12} remain in the cache because these two layers are played in requests from both high and low bandwidth clients, and therefore they are more popular. Similar to the previous experiment, required layers of the unpopular stream (s_{19}) are always played back from the server and stay in the cache only for a short period of time since they are not sufficiently popular. Therefore prefetching is never triggered for this stream.

Figure 10 shows average delivered quality of all streams over the entire experiment as a function of stream ID. Ideally, average delivered quality of each stream should monotonically decrease with stream popularity. However, Figure 10 does not illustrate such a characteristics. A closer examination of our results revealed that this behavior is caused by our LFU replacement algorithm. The choice of $\Delta = \infty$ implies that our replacement algorithm is a variant of the Least Frequently Used (LFU) algorithm. LFU-based algorithms have the well known anomaly that temporal distribution of requests plays a significant role in determining the popularities of the streams in the cache [9]. A less popular stream can stay in the cache for a long time if a large number of requests for this stream arrives early in the experiment. This suggests the use of limited values for Δ in order to age the stream popularities with time similar to LRU algorithm. We are currently investigating this issue.

5. RELATED WORK

Multimedia proxy caching is a new research area that has not been sufficiently explored. During recent years, a few commercial multimedia proxy caches have been developed [10, 11, 12]. While there is no technical information about these products, they apparently consist of a Web cache that is bundled with a media player. There are numerous works on proxy caching mechanism for Web objects (*e.g.*, [13, 14]). However, due to the larger size of multimedia streams compared to Web objects and streaming nature of delivery, existing proxy caching schemes seem to be inefficient for multimedia streams. The MiddleMan architecture [15] is a collection of cooperative proxy servers that collectively act as a video cache for a well-provisioned local network (e.g. Lan). Video streams are stored across multiple proxies where they

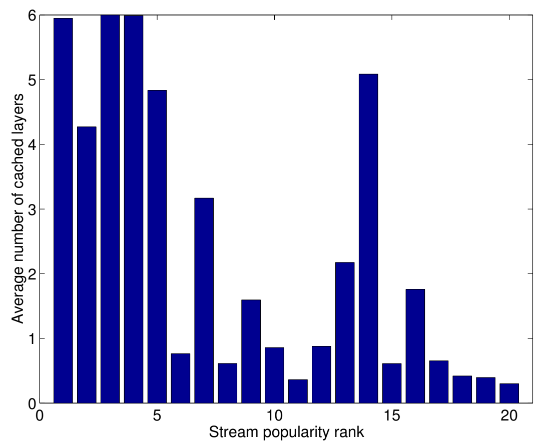

Figure 10: Cached stream quality as function of popularity

can be replaced at a granularity of a block. They examine performance of the MiddleMan architecture with different replacement policies.

A class of caching mechanisms for multimedia streams proposed to cache only selected portions of multimedia streams to improve delivered quality. Clearly, these solutions do not decrease the load on the server (or the network). To smooth out the playback of variable bit rate video streams, work in [16] proposes a technique called Video staging. The idea is to prefetch and store selected portion of video streams in a proxy to reduce burstiness of the stream during the playback. Sen et al. [17] also present a prefix caching mechanism to reduce startup latency. Work in [18] suggests caching only selective frames of a media stream based on the encoding properties of the video and client buffer size in order to improve robustness against network congestion. Work in [19] presents a caching architecture for multimedia streams, called SOCCER. SOCCER consists of a self-organizing and cooperative group of proxies. Work in [20] describes design and implementation issues of a single proxy in the SOCCER architecture that implements LRU replacement algorithm. They focus mostly on issues such as segmentation of multimedia streams and request aggregation. Work in [21] studies the layered video caching problem using an analytical revenue model based on a stochastic knapsack. The authors also develop several heuristics to decide which layers of which streams should be stored in the cache to maximize the accrued revenue.

In the context of media servers, various caching strategies of multimedia streams in main memory have been studied in prior work[22, 23]. The idea is to reduce disk access by grouping requests and retrieving a single stream to serve the entire group. Tewari et al. [24] present a disk-based cache replacement algorithm for heterogeneous data type, called Resource Based Caching(RBC). RBC considers the impact of resource requirement of each stream (*i.e.*, bandwidth and space) on cache replacement algorithms to maximize. Work in [25] further examined the RBC algorithm and presented a hybrid LFU/interval caching strategy.

Most of the previous work in this class treat multimedia streams similar to Web objects (*i.e.*, perform atomic replacement). Mocha complements previous work on multimedia proxy caching. More specifically, Mocha contributes the idea of quality adaptive caching.

6. CONCLUSIONS AND FUTURE WORK

This paper described the design and implementation of a quality adaptive multimedia proxy cache, called Mocha. We justified the need for quality adaptive caching of multimedia streams over the Internet and argued that layered encoding presents the most efficient approach to quality adaptive caching of multimedia streams. Mocha performs fine-grained prefetching mechanism and uses fine-grained replacement mechanism to maximize both delivered quality storage efficiency simultaneously. We presented Mocha's architecture and key components of our prototyped implementation on top of Squid. Our preliminary results show that Mocha can properly adapt the quality of a cached stream based on its popularity and on the available bandwidth between the proxy and interested clients. We also observed that LFU-based replacement algorithms are sensitive to temporal distribution of requests.

We plan to continue this work in a couple of directions. First, we are defining a new evaluation methodology for multimedia caches. The notion of delivered quality for cached streams reveals that traditional performance evaluation metrics (*e.g.*, Byte hit ratio) for Web caching are neither well-defined nor sufficient for evaluation of multimedia proxy caching mechanisms. Instead, performance evaluation of multimedia caching mechanisms should be examined along two dimensions 1) overall quality of delivered streams, and 2) ability of the cache in reducing the offered load to the network.

Second, we use this evaluation methodology to conduct exhaustive performance evaluation of fine-grained replacement and fine-grained prefetching mechanisms under more realistic workload and background network traffic. We also need to examine sensitivity of fine-grained replacement and fine-grained prefetching mechanisms to their main parameters. We also plan to compare performance of our proposed replacement algorithm with other proposed algorithms for multimedia caches in the literature. We plan to explore the effectiveness of off-line prefetching.

Finally, we plan to incorporate utility (*i.e.*, importance on perceived quality) of individual layers in replacement and prefetching mechanisms. Our current approach assumes a linear utility function where all layers result in similar improvement in perceived quality. However, most of the existing layered encoded streams exhibit a non-linear utility (*e.g.*, PSNR) behavior across different layers. The replacement and prefetching mechanism should consider both popularity and utility of a layer in order to minimize the load on the network while maximizing the overall delivered quality.

7. REFERENCES

[1] R. Rejaie, H. Yu, M. Handley, and D. Estrin, "Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet," in *Proceedings of the IEEE INFOCOM*, Tel-Aviv, Isreal, Mar. 2000.

[2] "Squid web proxy cache," http://www.squid-cache.org/.

[3] R. Rejaie, "An end-to-end architecture for quality adaptive streaming applications in the Internet,"

Ph.D. thesis, University of Southern California, SC/CS- Tech Report-99-718, Dec. 1999.

[4] H. Schulzrinne, A. Rao, and R. Lanphier, "RFC 2326: Real time streaming protocol (RTSP)," Apr. 1998.

[5] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet," in *Proc. IEEE Infocom*, New York, NY., Mar. 1999.

[6] R. Rejaie, M. Handley, and D. Estrin, "Quality adaptation for congestion controlled playback video over the Internet," in *Proceedings of the ACM SIGCOMM*, Cambridge, MA., Sept. 1999.

[7] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RFC 1889: RTP: A transport protocol for real-time applications," Jan. 1996.

[8] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T. Li, "Mitra: A continuous media server," *Multimedia Tools and Applications journal*, vol. 5, no. 1, July 1997.

[9] A. Silberschatz, J. Peterson, and P. Galvin, Eds., *Operating System Concepts*, Addison Wesley, 1992.

[10] Inktomi Inc., "Streaming media caching brief," 1998.

[11] "Infolibria MediaMall," 1999, http://www.infolibria.com.

[12] "RealSystem Proxy," http://www.realnetworks.com/.

[13] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Dec. 1997, pp. 193–206.

[14] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox, "Removal policies in network caches for world-wide web documents," in *Proceedings of the ACM SIGCOMM*, Stanford, CA., 1996, pp. 293–305.

[15] S. Acharya and B. C. Smith, "Middleman: A video caching proxy server," in *Workshop on Network and Operating System Support for Digital Audio and Video*, June 2000.

[16] 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 the IEEE INFOCOM*, Apr. 1998.

[17] S. Sen, J. Rexford, and D. Towsley, "Proxy prefix caching for multimedia streams," in *Proceedings of the IEEE INFOCOM*, 1999.

[18] Z. Miao and A. Ortega, "Proxy caching for efficient video services over the internet," in *9th International Packet Video Workshop (PVW '99)*, New York, Apr. 1999.

[19] M. Hofmann, E. Ng, K. Gue, S. Paul, and H. Zhang, "Caching techniques for streaming multimedia over the internet," Tech. Rep., Bell Laboratories, Apr. 1999.

[20] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul, "Design and implementation of a caching system for streaming media over the Internet," in *IEEE Real Time Technology and Applications Symposium*, June 2000.

[21] J. Kangasharju, F. Hartanto, M. Reisslein, and K. W. Ross, "Distributing layered encoded video through caches," in *Proceedings of IEEE Infocom*, Anchorage, AK, Apr. 2001.

[22] A. Dan and D. Sitaram, "Multimedia caching strategies for heterogeneous application and server environments," in *Multimedia Tools and Applications*, 1997, vol. 4, pp. 279–312.

[23] M. Kamath, K. Ramamritham, and D. Towsley, "Continuous media sharing in multimedia database systems," in *Proceedings of the 4th International Conference on Database Systems for Advanced Applications*, Apr. 1995.

[24] R. Tewari, H. Vin, A. Dan, and D. Sitaram, "Resource based caching for web servers," in *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, San Jose, CA., 1998.

[25] J. M. Almeida, D. L. Eager, and M. K. Vernon, "A hybrid caching strategy for streaming media files," in *Proceedings Multimedia Computing and Networking*, San Jose, CA., Jan. 2001.