AMPS: A Flexible, Scalable Proxy Testbed for Implementing Streaming Services [1]

Xiaolan Zhang, Michael K. Bradshaw, Yang Guo[§] , Bing Wang,
Jim Kurose, Prashant Shenoy, Don Towsley

University of Massachusetts
{ellenz,bradshaw,bing,
kurose,shenoy,towsley}@cs.umass.edu

The MathWorks
[§] yguo@mathworks.com

Abstract— **We present the design, implementation, and performance evaluation of AMPS — a flexible, scalable proxy testbed that supports a wide and extensible set of next-generation proxy streaming services. AMPS employs a modular architecture and is built on top of a commodity Linux system. We quantify the maximum achievable throughput for two of the principal components of the proxy - the control plane and data plane, and identify the CPU to be the system bottleneck. Through profiling studies, we further identify the kernel network protocol processing and the Network Reception Module inside the proxy to be the most CPU-intensive components. We also characterize the end-to-end performance along the server-to-proxy-to-client path. We discuss lessons learned and the various optimizations made in the course of our study to improve system performance.**

I. Introduction

The rapid growth of broadband users has led to a substantial increase in streaming media usage over the Internet. Proxies are commonly used, often in content distribution networks, to deliver high-quality streaming media to broadband users. While today's proxies support services such as caching and content forwarding, future proxies will support a wide variety of services such as content insertion, on-the-fly protocol and format translation, Tivo-like interactive operations, localized broadcasting (such as periodic broadcast [5, 10]), proxy prefix caching [25], proxy caching strategies [1, 6, 10, 22, 28], cooperating proxies [21], and streaming CDNs [27].

While there are a number of commercial (e.g., Darwin, RealServer, and Windows Media Server) and experimental streaming servers [2, 5, 8, 14, 16, 26], there is considerably less research on the design and implementation of streaming *proxies*. Existing proxy design and implementation work [3, 11, 13, 23] and commercial streaming proxies (e.g., RealProxy and Darwin) are primarily aimed at supporting stream reception/forwarding and/or a small set of proxy services, handling specific media formats and streaming protocols.

In the context of multimedia system research, there is considerable work on developing modular, reusable and composable platform for building multimedia systems. Examples include the Berkeley Continuous Media Toolkit [19], Dali multimedia software library [17], the open source project GStreamer [12], and commercial platforms such as Windows Media and RealMedia. These systems either have not considered the problem of supporting reusable multimedia components in the proxy setting, or are proprietary systems not publicly available for research purposes.

In this paper, we present the design, implementation, and evaluation of *AMPS* (Active Multimedia Proxy Services) — a flexible, scalable proxy research platform designed to support a wide, composable, and extensible set of next-generation streaming services. It is tailored for rapid prototyping of new multimedia protocols and proxy services. The platform's design is governed by two principles. First, it is highly modular with well-defined communication interfaces among modules so that all modules can be replaced and reordered to create new systems and/or services. Secondly, the platform is not tied to any signaling protocol, streaming protocol, or stream format. All signaling messages and multimedia streams, on entering the platform, are converted to the internal request protocol and stream format. This design feature enables us to support translation between different control signaling protocols and streaming formats [30].

Another important design goal of AMPS is scalability: the capability to handle high client request rates and to sustain high data throughput reliably. We have implemented AMPS in a commodity Linux system and studied the performance of the AMPS proxy over a switched-Gigabit LAN. We identify the CPU to be the bottleneck resource in the proxy. Through detailed profiling, we study the CPU load imposed by various system operations and proxy components. We also quantify the maximum achievable throughput, and characterize the end-to-end performance along the server-proxy-client path.

The remainder of this paper is organized as follows. Section II describes the architecture design and implementation of the AMPS platform. Section III discusses the experimental setup and performance metrics. Section IV presents our performance study results. Finally Section V concludes the paper.

II. Architecture

A. Architecture Overview

As shown in Figure 1, AMPS is composed of a collection of modules organized into three planes: the *service*

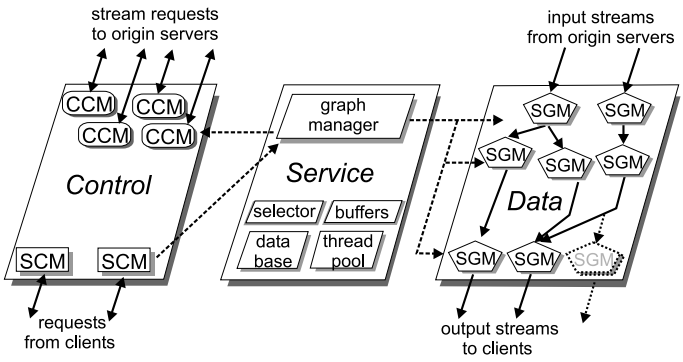

Fig. 1. The control, service and data planes in the AMPS proxy.

plane, the *control plane* and the *data plane*. The service plane provides system-wide services such as public services (database lookup, event dispatching), resource management (thread-pool, buffer manager), and a request-processing module known as *graph manager* (GM). The control plane is composed of *server control modules* (SCMs) and *client control modules* (CCMs). Each of these modules performs control signaling between the proxy and servers (from which the video is retrieved) and clients (to which the video is being delivered). Each control module communicates with external hosts, translating signaling messages into an internal format and passing stream requests to the graph manager in the service plane. The data plane consists of *stream graph modules* (SGMs) and *stream pipes* (henceforth simply referred to as pipes). Each SGM provides a specialized operation, taking zero or more streams as input and producing zero or more streams as output. A multimedia stream is abstracted into a stream of *frames* each identified by a *frame pointer*. Pipes pass streams by reference (i.e., the frame pointer) between SGMs. The *stream graph* represents the flow of streams among the SGMs in the data plane. In the stream graph, each SGM is represented as a node and each pipe is represented as an edge.

Figure 1 depicts how modules in an AMPS proxy interact with each other. Note that the proxy uses multiple SCMs, one for each protocol used by the proxy in proxy-client signaling. An SCM translates requests from the client into an internal format and passes the requests to the graph manager (GM) in the service plane. The GM serves requests by configuring the stream graph, choosing which SGMs to use and the order to connect them. In addition, the GM is able to fork the output of existing pipes to satisfy new requests (as illustrated in the lower right region of the data plane in the figure) without disrupting the service of existing streams. If the video needed to satisfy a client request is not locally available, the GM passes the request to the CCM that implements the signaling protocol of the origin server. The CCM negotiates with the server to receive the stream and informs the GM if the stream will be delivered and how the stream should be received.

B. Implementation

For the service plane, we have implemented all modules except the graph manager (which is the subject of our ongoing work). For the control plane, we have implemented an SCM and a CCM using the RTSP [24] protocol based on the *komproxyd* [3] RTSP parser. The RTSP SCM uses the service plane event-dispatcher to monitor all client TCP connections for new requests. Upon the arrival of a request, a thread from a thread-pool is dispatched to parse and process the request. For the data plane, we have implemented three SGMs, each run by a thread. The *Memory Loader Module* (ML) loads video data from disks and passes streams of frame pointers to the output pipes. Its design is derived from our previous multimedia streaming server work [5]. The *Network Reception Module* (NRM) receives streams from network and passes streams of frame pointers to the output pipes. The *Network Transmission Module* (NTM) retrieves streams from upstream pipes and transmits the packets to specified network addresses. To reduce context switches and thread overhead, there is one instance of each type of SGM in memory serving all streams.

For the performance study presented in this paper, we composed a streaming server, proxy, measuring client and workload generator from these modules, making use of a simple graph manager.

III. Experiment Setup and Performance Metrics

All experiments are performed on a Gigabit Ethernet LAN connected with a DELL PowerConnect 2508 8-port Gigabit switch. The proxy, server, workload generator and data sink applications are each run on a Dell OptiPlex GX260 (with a 1.8GHz P4 processor) running Redhat Linux 2.4.22. The proxy machine has 1GB RAM and two Intel Pro1000 MT Desktop Gigabit cards (the driver version is 5.2.20) connecting to a 33MHz/32bits PCI bridge. The server, data sink, and workload generator machine each have 512MB RAM and one NIC. The measuring client runs on a Dell OptiPlex GX1 (with a 448MHz P2 processor, 376MB RAM, and a 3Com 100Mbps NIC) running Linux 2.4.22.

Experiments are conducted using the following two configurations: a *signaling configuration* that is used to examine the performance of the control plane and the system as a whole, and a *data configuration* that is used to examine the throughput of the data plane. For both configurations, the server-proxy traffic and the proxy-client traffic are isolated on the two separate NICs in the proxy.

In the signaling configuration, the workload generator generates client session arrivals according to a Poisson process, with each session lasting a fixed amount of time (referred to as *client session duration*). The measuring client generates client

video file file	frame rate (fps)	bitrate (Kbps)	pkt rate (pkts/sec)
V10	10	12.5	10
V100	10	102.5	10
V300	15	307.4	30
V1024	30	1054	120

TABLE I
VIDEO FILES USED IN EXPERIMENTS

session arrivals in sequence, and logs the signaling delay and the reception times of data packets. Each client session (generated by the workload generator or the measuring client) requests a video located at the server through the proxy using the RTSP protocol. The *mute* mode of this configuration, in which no data streaming is performed, is used to exclusively study the control plane performance.

The data configuration consists of the server, proxy and datasink. The proxy internally simulates arrivals of stream requests, sends the requests via a single TCP connection to the server, receives data packets from the server, and dumps the data packets to the data sink.

The video files used in our experiments, listed in Table I, have different bit and frame rates meant to represent the range of video characteristics seen in practice. In each experiment, requests are for the same video, but the proxy treats each request equally by forwarding the streaming request to the server, and receiving and forwarding the video to the client.

We use *sar* and *netstat* to collect system resource usage and network performance at one second intervals, and call *ifconfig* and *ethtool* before and after each experiment to collect networking statistics on the proxy and server.

We report the system-wide *CPU usage* and *data throughput* of the proxy. As the proxy is the only application running on the system (except for Linux daemons), the system-wide CPU usage directly reflects the CPU overhead of the proxy. The data throughput is the amount of video data being received and forwarded by the proxy per unit time. For the end-to-end performance, the *frame inter-arrival time* and *signaling delay* observed by the measuring client are reported. Variability in the frame inter-arrival time reflects delay jitter experienced by the client. The signaling delay is the time from when the client sends a RTSP request to the time when it receives the response back from the proxy. It reflects the control signaling latency.

IV. Performance Results

In this section, we describe selected performance results. To understand the CPU usage within the proxy, we performed a profiling analysis on the proxy to characterize CPU usage of the various components in the proxy. We then report the system performance of the control plane, the data plane and the overall system. We close this section by presenting several design and implementation issues in the data plane modules and highlighting their performance impacts.

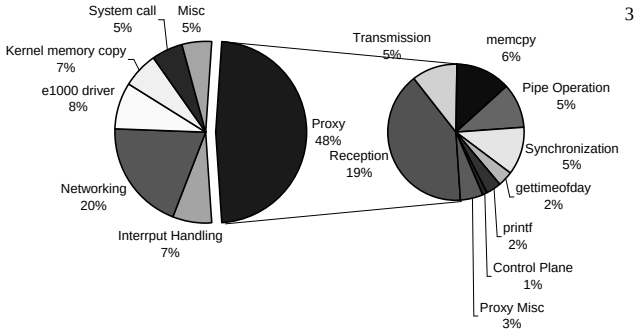

Fig. 2. Proxy CPU Profi ling

A. Proxy Profiling Analysis

We profiled the proxy using the OProfile [15] profiling system, where the timer interrupt is used to sample the PC (Program Counter) value. A summary is generated, in which the number of samples and the percentage for each routines (system/library/application) are reported. We ran an experiment using the signaling configuration. In the experiment, the workload generator generates client arrivals with a rate of 5 clients/second, where each client requests the V300 video, and have a client session duration of 120 seconds. Under this workload, the average number of sessions in the proxy is 600 and the average data plane throughput is 180 Mbps. The breakdown of CPU active time among various operations for the proxy is shown in Figure 2, where the left side pie chart shows the system level CPU time breakdown, and the right side pie chart further depicts the breakdown of CPU time within the proxy application. The percentage of CPU idle time (spent in *default_idle*) is 34%.

Compared to the profiling results for the streaming server reported in [14], where disk access was found to be the bottleneck, the streaming proxy shows a different system bottleneck. We observed that kernel network protocol processing takes up a significant percentage of CPU time (20%). This shows that fine tuning the network protocol stack for streaming workload (high data rate and a large number of sockets) is important to improve the proxy performance. As the proxy handles the reception and transmission of a large amount of network data packets, both the sending and receiving path of the network protocol processing are stressed. For example, the top routine in this category (accounting for over $1/5$ of the CPU time in this category) is *udp_v4_lookup_longway*. This routine looks up UDP socket for an incoming UDP packet, making use of a hash table of 128 entries that map destination port numbers to the socket structures.

Similar to the streaming server studied in [14], a substantial amount of CPU of the proxy system is spent in system calls (5%) and kernel memory copying (7%, due to the copying between user and kernel space). This demonstrates the overhead of user-kernel context switches, and the benefit of adopting schemes that eliminate data copying [9, 18].

Within the proxy application, the Network Reception Module (NRM) is the most CPU intensive (using 19% of the total CPU time). This is due to the functionality of NRM (i.e., receiving packets and introducing frames into pipes), and the complexity associated with handling potential packet reordering and losses for the incoming UDP streams. Streaming servers, which serve multimedia streams from local disks, don't need to handle these cases.

The profiling result also quantifies the overhead of using pipes to pass streams among SGMs. Pipe operations, which include querying the schedule of a pipe, popping/pushing frames into a pipe, takes up 5% of the CPU time. This is a reasonable amount of overhead to enable such high level modularization and data sharing. The AMPS proxy avoids the expensive memory copying operations by passing streams among SGMs by frame pointers. Each data packet is copied twice within the proxy: first by the NRM from a temporary packet receiving buffer to the internal buffer, and then by the NTM from the internal buffer to a temporary packet sending buffer. The overhead of user level memory copy (*memcpy*) is only 6% of the total CPU time.

The synchronization cost due to having multiple, yet a fixed number of, threads, including thread locking and sleeping operations, is 5%. Furthermore, the kernel process/thread scheduling overhead (which is counted as part of Misc) is around 0.09%. This suggests that adopting a single thread event-driven architecture probably would not yield a significant performance improvement.

Note that the overhead of the *gettimeofday* and *printf* system calls, which are mainly due to data logging (for performance evaluation purpose), is around 5%.

B. Performance Results Summary

In this section, we summarize the performance evaluation of the control plane, the data plane, and the overall system; detailed results can be found in [29].

For the control plane, we performed a set of experiments using the signaling configuration in mute mode with varying client arrival rate (with client duration of 120 second). Figure 3 plots the proxy CPU usage under various client arrival rates. The median and 95-th percentile of client-observed signaling delay are bounded by 56 ms and 120 ms, respectively, for all cases. Note that when the client arrival rate increases

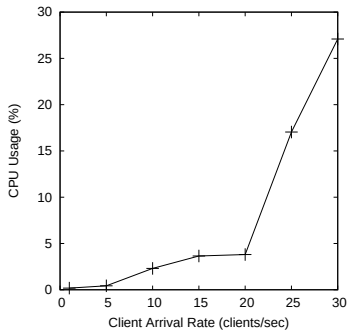

Fig. 3. Control plane CPU usage under various client arrival rates

video file	number of stream	bitrate (Mbps)	pkt rate (pkt/sec)
V10	2060	25.14	20,600
V100	1580	161.95	15,800
V300	710	213.13	21,300
V1024	280	288.2	33,600

TABLE II
MAXIMUM DATA PLANE THROUGHPUT

from 1 to 20 clients/sec, the CPU overhead increases only by 5%. In contrast, when the arrival rate inceases from 20 to 30 clients/sec, the CPU overhead increases by a factor of 6. Profiling analysis showed that this increase resulted from the increased overhead in TCP/IP processing and in the event-dispatcher that monitors all TCP connections.

For the data plane, we identified the maximum throughput achieved for different videos through experiments using the data configuration with the proxy configured to increase the number of streams until the proxy is saturated. Table II summarizes the maximum supported throughput for different videos. The proxy achieves maximum data throughput of 288.2 Mbps and 33,600 packets per second when the proxy receives and forwards a total of 280 V1024 video streams. The data throughput (bit and packet rate) sustained for lower rate videos is smaller due to the increased per-stream overhead.

We studied the overall system performance through the signaling configuration experiments with V300 videos. Client requests were generated at different rates, with each client requesting the first 120 seconds of the video. We found the proxy could sustain client arrival rates that result in an average of 600 simultaneous streams, generating an aggregate data throughput of 180Mbps. Analysis of client traces demonstrated that the proxy achieves smooth streaming under this load. Figure 4 plots the histogram of the frame inter-arrival time observed by the measuring clients. The mean of the frame inter-arrival time is 66.67 ms, which is consistent with the frame rate of the video (15fps). The maximum is 144.07 ms, within 2.2 times of the mean. We further analyzed the frame arrival time traces, and found that a pre-buffering of 52 ms at the client was enough to ensure smooth playback (no

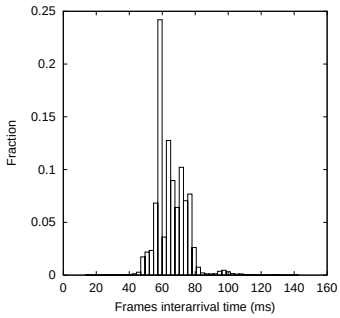

Fig. 4. Frame inter-arrival times at the client

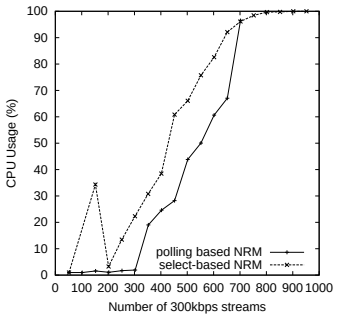

Fig. 5. Polling vs. *select*-based NRM

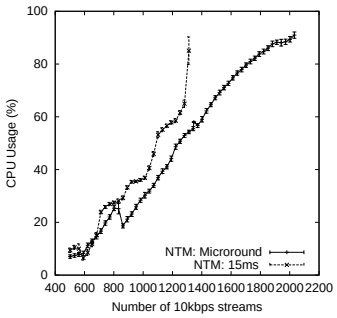

Fig. 6. CPU saving from the use of microrounds

buffer underflow).

C. Impact Of Various Design Decisions

In this section, we investigate design and tuning issues concerning two data plane modules within the proxy: the Network Reception Module (NRM) and the Network Transmission Module (NTM).

Network Reception Module Optimizations: The NRM operates in rounds with each round divided to two phases of operation: *(i)* receiving packets from the network and storing them in a staging area, and *(ii)* assembling frames from staged packets and putting the frames into the output pipes.

Our earlier profiling results indicated that the NRM is the most CPU-intensive component within the proxy. We have taken several approaches to speed up NRM operations. First, we designed a more efficient data structure for staging received packets. Secondly, we avoided the system overhead of memory allocation and deallocation by maintaining free lists for the constantly allocated/deallocated data structures. The third optimization is on the method with which the NRM performs the first phase operation. We originally made a *select* call on all receiving sockets to determine when packets were available (referred to as *select*-based NRM). Due to the well-known scalability problem of the *select* call [4, 7], we implemented another method that exploits the periodic nature of video data— the NRM simply reads each receiving socket in every round in a non-blocking fashion (referred to as polling-based NRM). We conducted the data configuration experiment requesting V300 videos to compare the efficiency of these two implementations. Figure 5 plots the CPU usage based on the number of streams that the proxy is serving. We observed that the polling method reduces CPU usage by as much as 30%, and when the number of streams is increased to above 700, the two methods achieve similar performance.

Network Transmission Module Tuning: The NTM also works in rounds. Suppose a round length of T_s ms is used. In each round, the NTM sends out all packets that are scheduled for delivery over the next T_s ms, and then goes to sleep for T_s ms. On wakeup, the NTM conducts the next round of transmission. Since the operating system provides no guarantee that the NTM will regain use of the processor at its requested time. the NTM detects when it has overslept and simply delays sleeping until it "catches up" with the delayed work.

Due to the 10 ms scheduling granularity of Linux 2.4.22, using a round length of less than 15 ms results in consistent oversleeping by the NTM thread. Given this constraint, we implemented two different solutions. In one implementation, we set the round length of the NTM to be 15 ms (the smallest sustainable round length). The other implementation uses the concept of a *microround*. In the microround scheme, the NTM uses a 33 ms round length and makes use of two 15 ms microrounds. The NTM serves half of the streams in each microround.

We conducted an experiment using the data configuration with V10 videos using these two schemes, and compared the proxy CPU usage of the proxy under the two schemes in Figure 6. The microround scheme provides a CPU saving of up to 30%, and is able to deliver one third more V10 streams than the round-based scheme. Under the same load, both schemes serve the same number of video streams at each 15 ms interval. The difference lies in the fact that the microround scheme only calculates the schedule and retrieves stream data for half of the clients, while the 15 ms round scheme performs the same operations (while handling a half of the video data) for all clients. This leads to a saving in CPU utilizing for schedule calculation and moving memory into the L1 (CPU) cache.

D. Other issues studied

In this section, we summarize additional experimental results. Interested readers should refer to the technical report [29] for more details.

- Substantial amount of system tuning is required to support a large number of client sessions and sustain the high data throughput reliably. This includes increasing per-process limits on the number of opened files, and various

buffer size in the networking stack [20].

- We have compared the system overhead and dispatching latency for several different implementations of the service plane event dispatcher. We found that traditional *select*-based implementation does not scale [4, 7], and a periodic polling-based implementation achieves up to 99% reduction in the CPU overhead, at the cost of the increased client signaling delay (by 30ms) - a price that is likely to be acceptable to streaming applications.
- We have studied the impact of our control plane threading model, showing that it provides a nice trade-off between efficiency and complexity.

V. Conclusions

We have designed and implemented a multimedia streaming research platform for supporting a wide range of proxy services. We evaluated our design and implementation through a series of experiments using a server/proxy/client configuration in a switched-Gigabit LAN setting.

We identified the CPU to be the bottleneck resource at the proxy, performed profiling to understand the overhead of various system components, and found the system network protocol processing and the NRM to be the most CPU-intensive components. The system performance studies showed that the control plane can handle a high arrival rate and a large number of concurrent client sessions, and quantified the maximum data throughput (up to 188.2 Mbps) that can be supported by the proxy's data plane. For the end-to-end performance, the proxy is able to provide good quality of service (as measured by delay jitter) to the client even under a relatively heavy load.

In addition to these performance results, we also learned several lessons from our prototyping efforts. First, system tuning is important to avoid configuration-induced system bottlenecks and packet loss. Secondly, to handle a large number of simultaneous sessions efficiently, efficient event (and packet) dispatching is needed; the periodic nature of multimedia stream can be exploited in the packet dispatching. Thirdly, the 10 ms Linux scheduling granularity must be taken into account for important system decisions, such as the choice of the round length.

Our ongoing work includes the implementation of the graph manager and several selected proxy services.

References

[1] J. M. Almeida, D. L. Eager, and M. K. Vernon. A hybrid caching strategy for streaming media files. In *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 2001.

[2] K. Almeroth and M. Ammar. An alternative paradigm for scalable on-demand applications: Evaluating and deploying the interactive multimedia jukebox. In *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, July/August 1999.

[3] Multimedia Communications Lab (KOM) at Darmstadt University of Technology. KOMproxyd open source project. http://dmz02.kom.etechnik.tu-darmstadt.de/KOMproxyd/.

[4] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[5] M. K. Bradshaw and et al. Periodic broadcast and patching services - implementation, measurement, and analysis in an Internet streaming video testbed. In *Proc. of ACM Multimedia System*, 2001.

[6] Y. Chae, K. Guo, M.Buddhikot, S. Suri, and E. Zegura. Silo, rainbow, and caching token: Schemes for scalable, fault tolerant stream caching. In *Journal of Selected Area in Communications*, 2000.

[7] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *USENIX Annual Technical Conference*, June 2001.

[8] S.F. Chang, A. Eleftheriadis, and D. Anastassiou. Developement of columbia's video on demand testbed. *Image Communication Journal:Special Issue on Video on Demand and Inerative TV*, 1996.

[9] P. Druschel. Operating systems support for highspeed networking. University of Arizona Ph.D. Dissertation CS-94-24, August 1994.

[10] D. Eager and M. Vernon. Dynamic skyscraper broadcasts for video-on-demand. In *Proc. 4^{th} Intl. Workshop on Multimedia Information Systems*, September 1998.

[11] S. Gruber, J. Rexford, and A. Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. *Computer Networks*, 1999.

[12] GStreamer Team. GStreamer: open source multimedia framework. http://www.gstreamer.net.

[13] V. Kahmann and L. Wolf. A proxy architecture for collaborative media streaming. In *Multimedia Systems*, December 2002.

[14] J. Lemon, Z. Wang, Z. Yang, and P. Cao. Stream Engine: A new kernel interface for high-performance internet streaming servers. In *Web Content Caching and Distribution Workshop (IWCW)*, 2003.

[15] J. Levon and et al. Oprofile. http://oprofile.sourceforge.net.

[16] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini multimedia storage server. *Multimedia Information Storage and Management*, 1996.

[17] W.-T. Ooi, B. Smith, S. Mukhopadhyay, H.H. Chan, S. Weiss, and M. Chiu. The Dali multimedia software library. In *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 1999.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. In *ACM Transactions on Computer Systems*, volume 18, pages 37–66, 2000.

[19] K. Patel and L.A. Rowe. Design and performance of the Berkeley Continuous Media Toolkit. In *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, February 1997.

[20] C. Perkins, L. Gharai, T. Lehman, and A. Mankin. Experiments with delivery of HDTV over IP networks. In *Proc. of the 12th Intl. Packet Video Workshop*, 2002.

[21] R. Rejaie and J. Kangasharju. Mocha: a quality adaptive multimedia proxy cache for Internet streaming. In *ACM Intl Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.

[22] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the Internet. In *INFOCOM*, 2000.

[23] S. Roy, J. Ankcorn, and S. Wee. Architecture of a modular streaming media server for content delivery networks. In *IEEE Intl. Conference on Multimedia and Expo*, 2003.

[24] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP), rfc 2326, April 1998.

[25] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proc. IEEE INFOCOM*, April 1999.

[26] M. Vernick, C. Venkatramini, and T. Chiueh. Adventures in building the stony brook video server. In *Proc. of ACM Multimedia*, 1996.

[27] B. Wang, S. Sen, M. Adler, and D. Towsley. Optimal proxy cache allocation for efficient streaming media distribution. In *INFOCOM*, 2002.

[28] 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 *Proc. IEEE INFOCOM*, April 1998.

[29] X. Zhang and et al. AMPS: A flexible, scalable proxy testbed for implementing streaming services. Technical Report 04-08, Department of Computer Science, University of Massachusetts Amherst, 2004.

[30] X. Zhang, D. Towsley, and J. Wileden. Towards interoperable multimedia streaming systems. In *Proc. of the 12th Intl. Packet Video Workshop*, 2002.