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

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

Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{ellenz,bradshaw,
bing, kurose, shenoy,towsley}@cs.umass.edu

[§] The MathWorks
Apple Hill Drive
Natick, MA 01760
yguo@mathworks.com

Abstract

We present the design, implementation, and evaluation of AMPS—a flexible, scalable proxy testbed designed to support a wide and extensible set of next-generation proxy streaming services. AMPS employs a modular architecture and is built using commodity hardware. We quantify the maximum achievable throughput, and identify the bottlenecks in the two main components of the proxy - the control plane and data planes. We discuss the impact of the proxy's threading architecture, timing and scheduling, and packets multiplexing/demultiplexing schemes. We also characterize the end-end performance along the server-to-proxy-to-client path, and proxy performance when implementing interval caching. We discuss lessons learned and the various optimizations made in the course of our study to improve system performance.

1 Introduction

The rapid growth of broadband users has led to a substantial growth 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, 11]), proxy prefix caching [34], proxy caching strategies [38, 11, 29, 2, 7], cooperating proxies [28], and streaming CDNs [37].

In this paper, we present the design, implementation, and evaluation of AMPS—a flexible, scalable proxy testbed designed to support such next-generation streaming services. AMPS employs a modular architecture to support a wide, composable, and extensible set of proxy services. There are a number of commercial (e.g., Darwin, RealServer, and Windows Media Server) and experimental streaming servers [3, 9, 6, 16, 21, 36, 5, 18], and considerable work on developing software libraries for building multimedia systems, including the Berkeley Continuous Media Toolkit [25], Dali Multimedia software library [24, 39], and the open source multimedia framework, GStreamer [35]. There has been considerably less research on the design and implementation of streaming *proxies*. There have been several proxy design and implementation works [15, 31] and commercial streaming proxies (such as RealProxy and the Darwin Streaming Proxy), but these efforts are primarily aimed at supporting simple stream reception/forwarding and/or a small set of specific proxy services. The goal of AMPS, by contrast, is to provide support for composable and extensible proxy services. The effort most closely related to our work is the *komproxy* open source project [22], which shares similar goals. The most important difference between our work and *komproxy* is our support for dynamic sharing of streams and translation among different control signaling protocols. We make use of the *komproxy* codebase in our implementation of the RTSP protocol and in our event-dispatcher.

In this paper we present a comprehensive performance study of the AMPS streaming proxy in a gigabit switched-LAN environment. We quantify the maximum achievable throughput, and identify the bottlenecks in the three main components of the proxy - the control plane, the service plane, and the data plane. We discuss the impact of the proxy's threading architecture, timing and scheduling, and packets multiplexing/demultiplexing schemes. We also characterize the end-end performance along the server-to-proxy-to-client path, and proxy performance when implementing interval caching. We discuss lesson learned and the various op-

*This work is done when Yang Guo was a postdoctoral research associate with the Dept. of Computer Science at UMass Amherst timizations made in the course of our study to improve system performance.

The remainder of this paper is organized as follows. Section 2 describes the architecture of the AMPS platform. Section 3 discusses the experimental setup and performance metrics of interest for our evaluation. Section 4 presents an empirical study of the performance of the AMPS proxy. Throughout section 4, we highlight lessons learned regarding the various alternatives we considered in designing the AMPS proxy. Section 5 demonstrates the composability of the AMPS platform through case studies. Finally in Section 6, we conclude the paper.

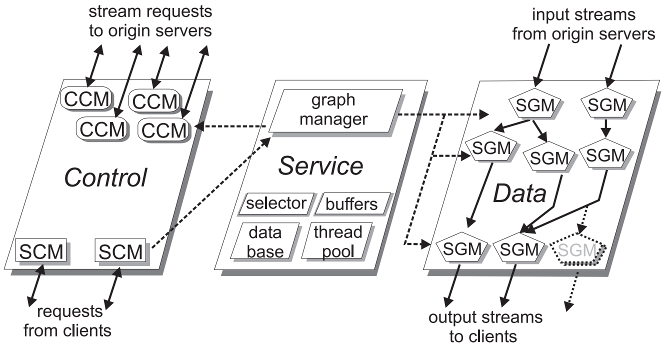


Figure 1: Modules interaction within AMPS Proxy to service a request

2 Architecture

AMPS (Active Multimedia Proxy Services) is a multimedia research platform that can be utilized as a streaming server, a client and (in the context of this paper) a proxy. AMPS is tailored for rapid prototyping of new multimedia protocols and services such as patching, transcoding, and picture-in-picture operations. The platform's design is governed by two principles.

- First, the platform is highly modularized. With the exception of communication structures passed between modules, all aspects of the platform are modules that can be replaced and reordered to create new systems and new services. Researchers can implement new algorithms through the reuse of existing modules.
- Second, the platform is not tied to any signaling protocol, streaming protocol, or stream format. All signaling messages and multimedia streams that enter the platform are converted to one internal request protocol and one internal stream format. These formats are flexible enough to support a wide range of signaling protocols and stream formats, while efficient enough to handle a heavy workload.

2.1 Architecture Overview

AMPS is composed of a collection of modules organized into three planes: the *service plane*, the *control plane* and the *data plane*. The service plane provides system-wide services such as database lookup, resource management, and a request-processing module known as the *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 video is retrieved) and clients (to which video is being delivered). Each control module communicates with external hosts, translating signaling messages into an internal format and passing stream requests/updates to the Graph Manager in the service plane. The data plane is composed 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. Pipes pass streams by reference between SGMs by abstracting multimedia streams into streams of frames. The *Stream Graph* represents the flow of frames among 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. Each SCM translates requests from the client into an internal format and passes the request to the Graph Manager in the service plane. The GM serves requests by configuring the stream graph, choosing which SGMs to use and the order in which to connect them. In addition, the GM is able to fork the output of existing pipes to satisfy new requests (as illustrated on the lower right region of the data plane in Figure 1).

If the video needed to satisfy a client request is not locally available, the GM passes the request to a 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. In the following subsections we describe the responsibilities and behaviors of the modules in each plane in more detail.

2.2 Service Plane

The service plane is responsible for all system-wide services and resources. The service plane is composed of resource managers (thread pools, buffer managers, and disk allocation), public services (database operations and event dispatching) and the Graph Manager.

The **Graph Manager (GM)** accepts a stream request from a Server Control Module and decides whether to satisfy or reject the request. To satisfy a request, the GM alters the Stream Graph, the representation of the flow of frames among SGMs in the data plane. The GM can add SGMs to receive streams, transcode streams, transmit streams, or combine SGMs to perform a higher level service. As there may be several ways that the GM can modify the Stream Graph (e.g., transcoding a locally stored video into the requested format or requesting the video from the server) to satisfy each request, the GM decides which solution is the most desirable for the system as a whole. In this paper, the GM is implemented to fulfill each type of request using a specific sequence of SGMs.

The **Selector** continuously monitors a set of file descriptors, typically a set of open sockets, referred to as the *interest set*. When a descriptor becomes readable, the selector calls the callback function associated with the descriptor. The callback function either services the file descriptor directly (e.g., accepts new TCP connections), or assigns a thread to service the request (e.g., parse and process a client request). File descriptors can be inserted into, or removed from, the interest set dynamically. There are different ways to monitor the interest set, as we will discuss in Section 4.2. Our *Selector* is derived from the komproxy [22], with added support for a variety of different methods to monitor the interest sets.

2.3 Control Plane

The control plane is responsible for signaling between the proxy and other entities such as servers and clients. The AMPS platform assumes that all communication is performed using a client/server paradigm. Control plane modules communicate with the GM through the use an internal data structure called *presentation tree*. The *presentation tree* is a tree structure consisting of nodes that represent streams or operations on streams (such as transcoding or merging). Each Server Control Module (SCM) fulfills the server functionality of a signaling protocol (e.g., RTSP [33], HTTP). The SCM translates stream requests into a presentation tree and sends the request to the GM. If the GM chooses to accept the request, the presentation tree is updated with information needed by the SCM to reply to the message (e.g., scheduling information and the specific modules used to satisfy the request). Each Client Control Module (CCM) provides the client functionality of a signaling protocol. CCMs are called by the GM to request streams from origin servers. The GM passes the requests in a presentation tree to a CCM. The CCM translates the request into the protocol of the server, and contacts the server. On the reception of the response, the CCM updates the presentation tree with the results and passes it back to the GM. Since all communication is modularized, not only with respect to the protocol used but also with respect to the client/server aspect, the proxy can "translate" requests from a client using one protocol to a server using another protocol [41].

2.3.1 Control Plane Implementation

We implemented an SCM and a CCM that speak RTSP based on the komproxy RTSP parser [22]. Both modules were implemented as a collection of functions that are executed by threads from a thread pool.

RTSP SCM. To handle a potentially large number of client connections simultaneously, the RTSP SCM uses the service plane *Selector* to monitor all client signaling channels (typically, TCP connections). When a message arrives at a connection, the selector sends the callback function registered with the socket to be serviced by the next available thread in the thread pool. The thread executes the callback function which receives, parses, and processes the incoming message. The thread is released after it has finished servicing the message. Note that a single RTSP connection can be serviced by several threads over the lifetime of the session. By releasing threads between incoming messages, the platform uses fewer threads than a thread-per-client model. We refer to this thread model as a *thread-per-request* model.

RTSP CCM. Unlike the RTSP SCM, the RTSP CCM does not release the thread when communicating with servers. This design requires more threads, but greatly decreases the implementation's complexity. We will discuss the repercussions of this decision in Section 4.3.

2.4 Data Plane

The data plane is responsible for all multimedia streams entering, exiting, generated by, and consumed by the AMPS platform. The data plane is composed of Stream Graph Modules that consume and produce streams and Stream Graph Pipes that pass streams among SGMs. To ensure that the data plane can operate with all stream formats, all incoming streams are converted to an abstract stream format. When a stream enters (or is generated by) the proxy through a SGM, the SGM divides the stream into *frames* and places the frames in memory. A frame is an abstract data structure that contains a unit of stream data, defined for each underlying multimedia stream format. For most video stream formats, a frame is equivalent to a video frame. AMPS represents a frame using a *frame pointer*, a data structure that locates the stream data within a frame.

2.4.1 Data Plane Implementation

An SGM is a stream filter that consumes zero or more input streams and produces zero or more output streams. Each SGM executes on a separate thread. As a series of SGMs must be used to satisfy a request, each SGM adheres to a delivery schedule for each stream that it outputs. An SGM meets deadlines by waking up periodically to produce the segments of the streams whose deadlines are due before it wakes up again. Each waking period is referred to as a *round* and the length of time between waking periods is referred to as the *round length*. To reduce the number of context switches and thread overhead, there is one instance of each type of SGM in memory, with this one instance serving all streams that use this type of SGM service.

Stream Graph Pipes are data structures used to pass frames from an upstream SGM to one or more downstream SGMs. In order to limit data copying in the application space, pipes use frame pointers instead of moving the stream itself. A pipe with n downstream SGMs simulates n queues. When the upstream SGM pushes a frame pointer into the pipe, the pipe places the frame pointer into each of the n queues. Each of the n downstream SGMs pops frame pointers from its respective queue. In implementation, a pipe simply stores each frame pointer until all downstream modules have retrieved it.

When no downstream SGM is interested in the stream passing through a pipe, the pipe prunes itself. The pipe first calls the callback functions registered with the pipe to notify the control plane that the pipe is no longer in use. The pipe then notifies the upstream SGM that the stream is no longer needed. If the SGM determines that it consequently no longer needs an input stream, the SGM unregisters its interest in the pipe that delivers the stream. This process is used to recursively remove unused streams from the system.

We have implemented three SGMS in the AMPS proxy studied in this paper:

The **Memory Loader Module (MLM)** loads pre-packetized video data from disk, and passes a stream of frame pointers to output pipes. Its design is derived from our previous work on streaming multimedia servers [5].

The **Network Reception Module (NRM)** reads streams from the network and passes streams of frame pointers to outbound pipes in a round-based fashion. During every round, the NRM performs a two-phase operation. In the first phase, the NRM receives packets from the network and stores them in a staging area. In the second phase, the NRM calculates the frames that need to be sent in the next 33ms for each stream, fetches these packets from the staging area, and assembles them into frames to be put into the corresponding output pipe. If packets are not received by this time, a missing or incomplete frame message will be put into the pipe. If these packets are received later by the NRM, they will be dropped on reception. We will examine the consequences of this NRM design in Section 4.6.

The **Network Transmission Module (NTM)** retrieves streams from upstream pipes and transmits the packets to specified network addresses. Ideally, the NTM should smoothly transmit a stream by spacing output packets evenly at a constant bit rate. However, doing so for a large number of streams places a burden on the CPU, and the NTM must share the CPU with other proxy functions. The NTM thus sleeps until the start of a round, wakes up, and sends all of the packets scheduled for delivery before the next round, and returns to sleep. The length of time that the NTM sleeps is very small. Since there is no real-time support in the operating system, there is no guarantee that the NTM will regain use of the processor at its next requested wakeup time. Thus, the NTM must detect when it has overslept and simply not sleep until it "catches up" with the delayed work. If the round length is smaller than the frame time of a stream, the NTM sends a portion of the frame. This allows traffic smoothing at finer granularities. In Section 4.7 we describe the tradeoffs in choosing the round length for NTM.

3 Experimental Setup and Performance Metrics

To evaluate the design of our AMPS proxy, we perform a set of experiments in a LAN environment to identify system bottlenecks, evaluate design alternatives, and study how proxy performance scales with increases in demands.

3.1 Experimental Setup

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 running Redhat Linux 2.4.22 with a P4 1.8-GHz processor. The proxy machine has 1GB RAM and two Intel Pro 1000 MT Desktop Gigabit cards (Intel Corp. 82540EM Gigabit Ethernet Con-

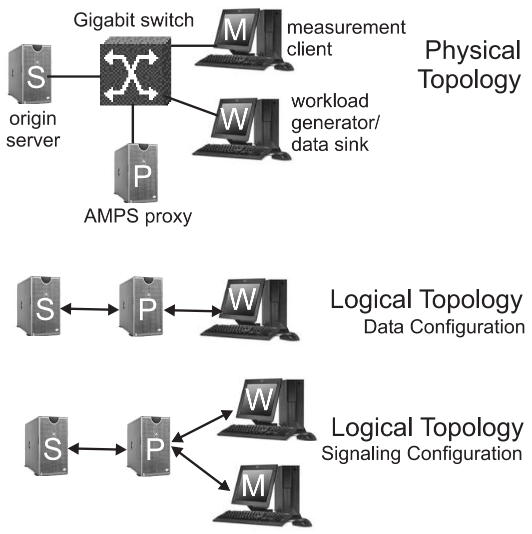


Figure 2: xperimental Configurations

troller) connecting to a 33MHz/32bits PCI bridge The server, data sink, and workload generator machine have the same hardware as the proxy except that they have 512MB RAM and one NIC each. We use the 5.2.20 version of the e1000 driver for the Intel Pro 1000 NICs. The measuring client runs on a Dell OptiPlex GX1 with a P2 448 MHz CPU and 376M RAM, running Linux 2.4.22, with a 3Com 100Mbps NIC.

Experiments are conducted using one of two configurations, as shown in Figure 2. For both configurations, server-proxy traffic and proxy-datasink/workload generator/client traffic are isolated on the two separate NICs in the proxy.

The *data configuration* setup shown in Figure 2 is used to examine the maximum throughput of the proxy's data plane by removing the proxy's client signaling component. The proxy internally simulates arrivals of stream request. Each simulated request prompts the proxy to send a stream request to the server. The server accepts the request and sends back a response. On receiving the response, the proxy receives the stream from the server and forwards it to the data sink. The data sink opens a set of UDP sockets and binds them over a range of port numbers. (The use of an explicit data sink prevents the proxy from receiving ICMP port-unreachable messages.) In the data configuration, the proxy first initiates a large set of stream to place load on the system, and then periodically adds new streams until the proxy is saturates. Each requested video is longer than the duration of the experiment to ensure that the load on the proxy does not decrease.

The *signaling configuration* setup shown in Figure 2 is used to examine the performance of the control plane and the system as a whole. The workload generator generates client

video file	**frame rate**	**bitrate (kbps)**	**pkts in frame**	**pkts/ sec**
V10	10	12.5	160Byte	10
V100	10	102.5	1312Byte	10
V300	15	307.4	1448B, 1208B	30
V1024	30	1054	1448B,1448B, 1448B, 153B	120

Table 1: Video files used in experiments

session arrivals according to a Poisson process, with each session lasting a fixed amount of time (referred to as *client session duration*). By varying the client arrival rate and client session duration, the workload generator can simulate different workloads. The measuring client generates client 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) opens a TCP signaling connection with the proxy, negotiates and initiates the delivery of a video through RTSP, and at the end of the session, tears down the session and closes the TCP connection. In the *signaling configuration* setup, the server and proxy support a *mute* mode, in which no data streaming is performed; this allows us to isolate and exclusively study control plane performance.

We use a set of constant bit rate videos for our experiments. The video files, listed in Table 1, have different bit and frame rates meant to represent the range of video characteristics seen in practice. In each experiment, requests are made for the same video, but the proxy does not perform caching and treats each request separately (except for the performance study using interval caching reported in Section 5.2), i.e., the proxy forwards the streaming request to the server, and receives and forwards the stream to the client. As the proxy throughput is our main focus, such simple workload generation suffices.

3.2 Performance Metrics

We make use of several monitoring tools to evaluate system performance. We use *sar* [13] and *netstat* to collect system resource usage and network performance at one second intervals. Networking statistics on the proxy and server are collected over the course of each experiment using *ifconfig* and *ethtool*. We profile the proxy using OProfile [19].

We will report on the following proxy performance measures:

CPU Usage: The system-wide CPU usage is reported. As the proxy is the only application running on the system (except for ordinary Linux daemons), this value directly reflects the CPU usage of the proxy.

Data Reception Throughput: This is the amount of video data being received by the proxy per unit time.

Data Forwarding Throughput: This is the amount of video data being forwarded to clients by the proxy per unit time.

On the client side, we are interested in the following metrics that reflect viewing quality:

Frame Interarrival Time: This is the difference between the arrival times of consecutive frames. Variability in the frame interarrival time reflects jitter experienced by the client.

Startup Latency: This is the time from when the client first sends an RTSP *PLAY* request to the time at which it receives the first video data packet.

Signaling Delay: This is the time from when the client first sends a RTSP request to the time at which it receives the response back from the proxy.

4 Performance Results

In this section, we first discuss the tuning of various system parameters. We then study the impact of the control plane threading model, compare several different implementation options for the selector, and report on the performance of the control plane. For the data plane, we first compare the performance of different Network Reception Module design choices, discuss the tuning of the Network Transmission Module, and then examine the maximum supportable throughput of the data plane. Finally, we report on the performance of the combined control and data planes, and study the end-end performance observed by the clients.

4.1 System Tuning

In this section, we discuss the lessons learned about system tuning, and how various system parameters should be set in order to improve proxy performance.

The nature of proxy workload makes its performance greatly dependent on the OS networking stack. We observed substantial performance improvement after we switched from the Linux 2.4.18 kernel to the 2.4.22 kernel. This is possibly due to the NAPI driver model [32, 30] introduced to Linux since version 2.4.19. This model allows the CPU disable receive interrupts while polling the NIC card, and thus dramatically decreases the receive interrupt rate (and consequently the CPU utilization) at high packet arrival rates.

The NIC driver e1000 provides several options for tuning the receiving/transmitting interrupt rate generated by the card. These settings affect the CPU usage, latency, and packet loss at the proxy. As the settings of these parameters that achieve best performance is workload dependent (more specifically, packet-arrival-rate dependent), we used the default settings.

We found that several default settings for current Linux distribution were not well-suited for the workload of a multimedia proxy, resulting in excessive dropping of clients and network packets, even when the CPU was not fully utilized. Our first goal was to increase the number of sockets that the proxy can simultaneously keep open. This problem is commonly addressed in web servers and has well-known solutions [1]. We increased the maximum number of file handles that a process can maintain from the default value of 1024 to 8192. We also increased the OS limit on the number of half-open TCP connections to allow the proxy to handle bursty client TCP connection arrivals.

Using the default Linux settings, we measured significant losses in the server-to-proxy path at relatively low bandwidth utilizations, even if we directly connected the proxy and the server with a cable. To understand why packets were being dropped, we studied the Linux networking protocol implementation [30, 14], and found that the default settings were not designed to handle constant high throughput as is needed for multimedia systems. We found that the suggested settings for several variables concerning buffering at various levels in the network stack needed to be adjusted [26]:

qdisc (queuing discipline) length at the sending NIC: the default value is 1000 in Linux 2.4.22. We set it to 1,000,000 for sending NICs.

TxRing at the sending NIC: the default value is 1024 packets. We enlarged this to 2048 for sending NICs.

RxRing at the receiving NIC: the default value is 256 packets. We enlarged this to 2048 for receiving NICs.

Socket Receiving Buffer Size: the default size is 64KB. We set it to be large enough to buffer one second of the video at the proxy.

After these changes, we were able to increase the data throughput between the server and proxy until the proxy's CPU was saturated, with all losses (a negligible amount) occurring at the proxy's receiving NIC (the *rx_ovr* error reported by *netstat*). We will further discuss loss behavior in Section 4.8.

Lesson : To handle a high volume of network traffic, it is important to tune the buffer sizes

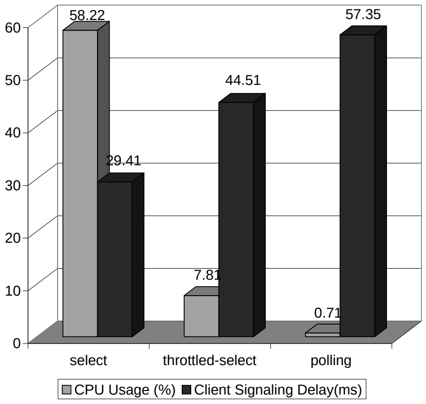


Figure 3: CPU Usage using different Selectors

used at various levels of the network protocol stack for both sending and receiving. The buffer needs to be large enough to absorb reasonable bursts of traffic, as well as delays in processing data.

4.2 Impact of Different Event-dispatching Implemenations

Recall that our implementation of the RTSP SCM makes use of the service plane *selector* to monitor the TCP connections with all clients. The main functionality of the selector is to monitor an interest set (more specifically, TCP sockets) for packet arrivals. As we'll see, the method by which the selector monitors the interest set greatly influences the performance of the control plane (and therefore, the system).

We conducted a set of experiments using the *signaling configuration* (as shown in Figure 2) in *mute* mode with a mean client arrival rate of 20 arrivals per second, and a client session duration of 100 seconds. In this setting, there are an average of 2000 simultaneous client sessions in the proxy. We plot proxy CPU usage and client signaling delay using several selector implementations in Figure 3.

We first consider the use of the *select* system call to monitor the interest set. In this approach, the selector thread runs in rounds that start by initializing the interest sets, then calling *select* with the interest sets, checking for ready file descriptors and then calling the associated callback functions. After serving all ready file descriptors, the thread proceeds to the next round. This baseline case utilized 58% of the CPU. Given the space limitations of this paper, we refer the interested reader to [40] and [8] for additional discussion of implementation and scalability issues associated with the *select* system call.

Our next approach improves upon the previous approach by adding a sleeping phase of 15ms at the end of each round. We call this approach the *throttled selector* approach. This approach decreases the frequency with which *select* is called, thus allowing more file descriptors to become readable at each round. Therefore, the cost of initializing interest sets and performing the *select* call is amortized over the multiple ready file descriptors being selected. Experiments showed that CPU usage decreased to 7.8%, at the cost of longer signaling delays experienced by client. However, the delay is only 44.5 ms (and the added delay is necessarily less than 15 ms) and is thus not a significant deterrent to using a throttled selector.

Our final experiment was to implement the selector using a simple polling mechanism. In this approach, the selector directly checked each of the file descriptors to see if it was readable (using the *recv* system call with *MSG_PEEK* option to peek at one byte of data from the socket), and called the associated callback functions if there was data to be read. The selector slept for 15 ms between pollings. This approach reduced CPU usage to 0.71% (which is a 99% drop from baseline case) at the cost of a factor of two increase in client signaling delay. We found that the savings in CPU utilization justifies the increase in delay of 28ms. For all remaining experiments we make use of this polling-based selector.

Several past works have studied the use of the *select* (and similarly, *poll*) system call as the event-notification mechanism in the context of web servers, and/or tried to provide more scalable event-notification mechanisms [27, 4, 8]. For example, the Posix RT Signal mechanism supported in Linux has been shown to scale well. We chose not to use this mechanism because it requires the application to handle RT signal queue overflow, which increases the design and implementation complexity. Two new mechanisms recently introduced into Linux are the */dev/poll* and *epoll* [20], which have similar functionalities as the *select* and *poll* system calls. We didn't make use of these calls as they were not incorporated into the Linux kernel (2.4.18) with which we started our implementation. It would be interesting to study the performance of the selector implemented using these new mechanisms.

Lesson: Polling improves efficiency of the event-dispatcher at a minimal cost of increased dispatching delay.

4.3 Impact of control plane threading model

As described in Section 2.3.1, we employ a thread-per-request model to implement our client-to-proxy RTSP SCM. If the proxy needs to contact an origin server, calls are made to the RTSP CCM, which holds the thread while waiting for a response from the server. In this section, we evaluate the im-

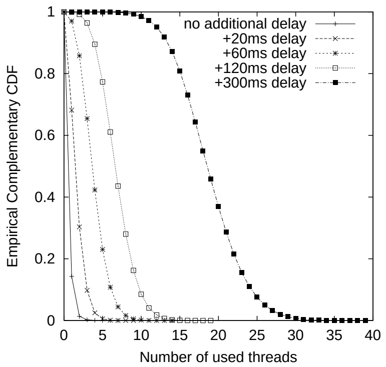


Figure 4: Threads Usage under various server-proxy signaling delay

pact of this threading model through experiments and modeling.

We define the *server-proxy signaling delay* to be the time from when the proxy sends a request to an origin server until the time it receives a response. This delay affects the holding time of the thread and therefore the total number of threads used in the control plane. We measured the server-proxy signaling delay in our testing environment to be less than 10ms. To simulate a longer server-proxy signaling delay, we instrumented the proxy so that additional delay could be added. We then conducted a set of experiments using the *Signaling Configuration* in *mute* mode with a client arrival rate of 20 per second, and a client session duration of 10 seconds. The proxy was initialized with a thread pool of 40 threads. We added a delay varying from 20 ms to 300 ms to the server-proxy signaling delay, and logged the usage of threads in the proxy. Figure 4 plots the empirical CCDF for the number of used threads in the thread pool for different delays. The results show that for a delay of less than 120 ms, a thread pool of 15 threads would be enough to guarantee no thread queuing delay. Even for a large server-proxy delay delay of 300 ms (e.g., that might be observed in a transcontinental path), a thread pool of 25 threads is sufficient to ensure that 95% of the client requests experience no thread queuing delay.

In contrast, under this workload, the thread-per-client model would require an average of 200 threads, as on average, there are 200 simultaneous client sessions in the system. Furthermore, the number of threads required using our model does not increase as the client session duration increases. The thread-per-client model, on the other hand, would require more threads as the client session duration increases.

We can model the threading behavior of the thread-per-client model as an $M/G/\infty$ queue with a service time equal to the server-proxy signaling delay. The prediction of thread usage from this model is very close to the experimental results. In an actual system, where client arrival rate and server-proxy signaling delay are not known *a priori* and can be dynamically changing, we can adapt the thread pool size based on current thread pool usage and system load.

arrival rate	CPU Usage(%)	client signaling delay	
		median	95 percentile
1	0.076 ± 0.09	31ms	103ms
5	0.188 ± 0.23	35ms	103ms
20	4.67 ± 0.61	56.8ms	120ms
40	26.76 ± 1.28	55ms	141ms

Table 2: Control Plane Performance

Lesson : To handle a large number of simultaneous clients, the concurrency model needs to be scalable. We employ an event-dispatcher and a thread pool to implement a thread-per-request model for the control plane.

4.4 Control Plane Performance

We used the *Signaling Configuration* in *mute* mode to study the performance of the control plane. The workload generator generated client arrivals at rates of 1, 5, 20, and 40 client arrivals/second, with each client having a duration of 120 seconds. The CPU usage and client observed signaling delay are summarized in Table 2.

The control plane handles very high client arrival rates (up to 40 clients per second) and large numbers of simultaneous client sessions (up to 4800) without incurring large CPU overheads. In general, the control plane is not the most resource-consuming component of the system, one can expect that with the deployment of services such as prefix caching and periodic broadcast, the performance of the control plane will become more important.

4.5 Fluctuations in CPU Usage for Data Plane Experiments

In the following three sections we will investgate several facets of the data plane. We used the *data configuration*(as shown in Figure 2) for all the experiments reported. While the load increases linearly in this configuration, we observed certain unexplained CPU usage fluctuated as shown in Figure 5, under a load of 150 to 300 streams using select, and under a load of 380 to 420 streams using polling. Figures 6 and 7 reveal similar artifacts. The fluctuations reappear at

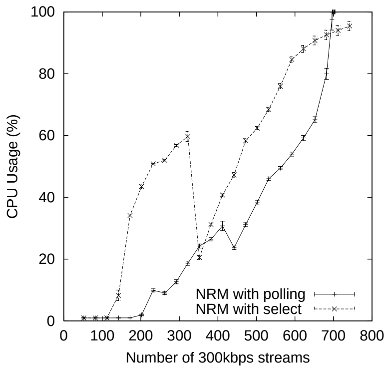


Figure 5: Polling vs select based Reception Module

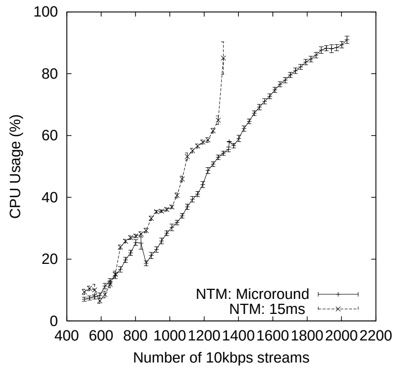


Figure 6: CPU saving of NTM Microround scheme

different times during repeated trials of the same experiments that involve the data configuration. Experiments run in Linux kernel 2.4.18 did not show this behavior, but had significantly lower overall performance.

To analyze the cause of these fluctuations, we profiled the system during experiments that used the data configuration. The profiling information was saved for each 100 second epoch of the experiment. We then compared the profile during the peak of the fluctuations and then again just after the peak had occurred.

When a polling-based NRM is used, we discovered that the increase in CPU usage resulted from an increase in time spent performing interrupts, in softirq handling, and in the e1000 driver routines that allocate receive buffers, cleanup buffers and enables receive interrupts. This effect was not due to an increase in the rate of interrupts, as the interrupt rate did not experience such fluctuations. When a select-based NRM is used, profiling results did not offer a strong candidate for the increased CPU usage. We suspect it is due to caching but leave further consideration of these fluctuations for future work and focus here on the trend revealed by the plots in the remainder of this paper.

4.6 Network Reception Module

Recall that the NRM described in Section 2.4.1 operates in 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. The method with which the the NRM performs the first phase operation can dramatically affect system performance. We investigated two different ways to determine when to receive packets from the network. Our first implementation employed a *select* call on all receiving sockets to determine when packets were available. The second implementation assumes that packets will arrive at the proxy at a regular rate and simply checks for new packets every 15 ms. We refer to this technique as polling (not to be confused with the *poll* system call).

To compare the efficiency of these two implementations, we used the *Data Configuration* requesting V300 videos. Figure 5 displays the CPU utilization based on the number of streams that the proxy is serving. We found that the polling technique reduces CPU usages by as much as 30%.

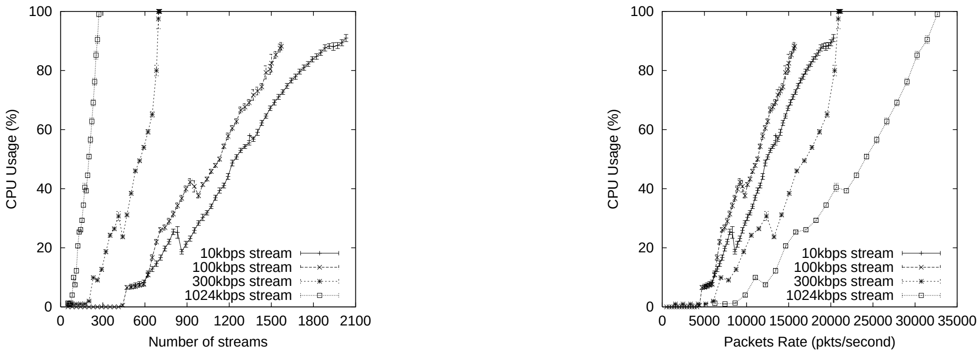
Lesson:Due to the periodic nature of multimedia streams, significant resource savings can be achieved by periodically scheduling reception of network packets instead of using select to see if packets have arrived.

4.7 Network Transmission Module

Recall that the NTM is round-based. 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.

Linux 2.4.22 supports a 10ms scheduling granularity. As a result, we found that using a round length of less than 15 ms results in consistent oversleeping by the NTM thread, as it never wakes up before the next deadline.

Given these constraints, 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 16 ms microrounds. The NTM serves half of the streams in each microround. To compare the CPU



(a) Proxy CPU usage vs number of streams

(b) Proxy CPU usage vs packets rate

Figure 7: Scalability of Data Plane

utilization of these two schemes, we conducted an experiment using the *Data Configuration* with V10 videos. Figure 6 compares the CPU usage of the proxy for the two schemes. For small numbers of videos, both schemes place similar loads on the proxy. However, as the number of streams increases, the microround scheme provides a savings of up to 30%. The microround scheme is also able to deliver one third more streams than the round-based scheme. For both schemes, the throughput is the same, however the microround scheme only calculates schedules and retrieves stream data for half of the clients. This leads to a savings in utilizing the processor for calculation and for moving memory into the L1 cache.

Lesson: It is necessary to consider the timing and scheduling effects caused by the 10 ms scheduling granularity of Linux. This limits amount of smoothing of outgoing traffic. Batching work more efficiently and working in microrounds improve performance.

4.8 Data Plane Throughput

In this section we report the maximum throughput of the proxy. We used the *Data Configuration* with different video files, where the proxy increased the number of streams until the proxy was saturated. Figure 7(a) plots the relationship between the proxy CPU utilization and the number of videos the proxy receives and forwards, while Figure 7(b) plots the relationship between the proxy CPU utilization and the number of packets received by the proxy at each second. Table 3 summarizes the maximum number of video streams and data throughput that the proxy can support for different videos.

The results show that while proxy CPU usage increases (as expected) with an increasing data rate, the number of videos being handled also influences utilization. That is, for the same overall data rate, the configurations with a smaller number of (higher rate) videos had lower CPU utilization than configurations with a larger number of lower bitrate videos. This demonstrates that incurred overhead also depends on the *number* of streams being handled.

video	max streams	bitrate	pkt rate
V10	2060	25.14mbps	20,600
V100	1580	161.95mbps	15,800
V300	710	213.13mbps	21,300
V1024	280	288.2mbps	33,600

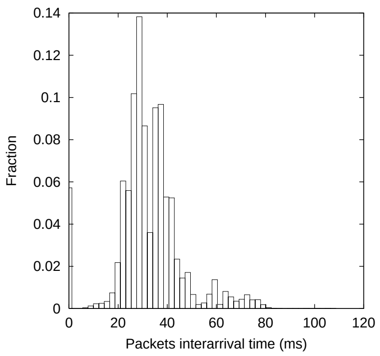
Table 3: Maximum Data Plane Throughput

Previous work [12, 23] has revealed the effect of packet spacing on the experienced loss rate over an Internet path. The work reported in [23] also revealed that enlarging the Inter Packet Gap (IPG, a MAC layer parameter) of the sender's Gigabit card can decrease the packet loss rate. Our experimentation has shown that even for a single-hop gigabit path, packet spacing is still necessary to prevent packets loss due to receiver buffer overflow. For example, by introducing delay between packets transmitted by the server (i.e., smoothing out server transmissions), the proxy experienced very low loss. For experiments with video V10 and V100, the proxy experienced no loss. For experiments with video V300 and V1024, the proxy experienced loss only after the packet arrival rate reached 20,000 packets per second. All losses occurred at the receiving NIC of the proxy, due to receiving buffer overflow at the NIC. The loss rate was below 0.1% per second, typically in bursts.

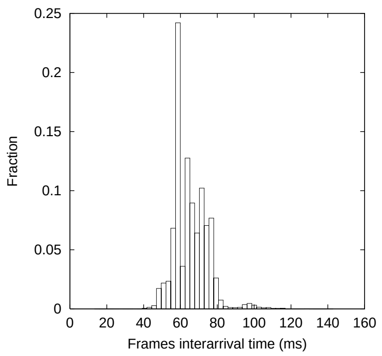
Observation: After tuning buffer sizes and smoothing out server transmissions, the CPU becomes the system bottleneck at the proxy.

client arrival rate	CPU Usage (%) and 95% CI			Synchronization Overhead	client startup latency (sec)
	Data only	Control only	System		
1	1.99 ± 0.21	0.076 ± 0.09	3.205 ± 0.392	1.139	4.18 ± 0.007
5	53.97 ± 0.73	0.188 ± 0.23	56.00 ± 0.55	1.842	4.18 ± 0.004

Table 4: Overall System Performance



(a) histogram packet interarrival time



(b) frame interarrival time

Figure 8: Client Reception Quality with client arrival rate of 5

4.9 Overall System Performance

We first examined the overhead introduced by the interaction between the control plane and the data plane by comparing their separate CPU usages with the system CPU usage. We then analyzed the video delivering quality as observed by measuring clients.

We measured the system's overhead by using the *Signaling Configuration* with V300 videos. Client requests were generated at different rates, with each client requesting only the first 120 seconds of the video. Table 4 summarized the CPU usage and client startup latency under different client arrival rates. We find that for arrival rates larger than 5 requests/second, the proxy's CPU is unable to meet the demand.

We see that there is a small cost in synchronization between the control plane and data plane (1.129% at a low load and 1.942% at high load). We also see that the client startup latency remains constant, indicating that there is not a decrease in proxy responsiveness as the load increases.

We analyzed client-side reception quality when the workload generator created five clients per second. Figure 8 plots the histogram of the client packet- and frame-interarrival time. The variance of the frame interarrival time is 84.785 ms^2, the standard deviation is 9.207 ms, and the coefficient of variation is 0.138. The maximum jitter is 144.07 ms, and is within 2.2 times of the mean 66.67 ms. The average client startup latency is 4.18 seconds. Most of this time can be attributed to the manner in which the proxy and the server interact. There is a two second latency at the proxy. This is used to eliminate the jitter between the proxy and the server. The server on receiving a request for a stream, will commit itself to deliver the stream starting in the next second. This is necessary due to the one second round length used by the Memory Loader.

All observations at the measuring client observed no loss. However, there are frequent pair-wise packet reordering occurring between pairs of packets within the same frame. This reordering doesn't affect the viewing quality of the client. Similar phenomena have been reported by [26], where they conjectured that a race condition in Linux kernel resulted in the reordering.

4.10 Profiling Results

To understand the CPU usage of the proxy, we profiled the system, and analyzed the breakdown of CPU time.

Figure 9 plots the breakdown of CPU time for the proxy using the *data configuration* setup with a fixed number(600) of v300 streams. Figure 10 plots the breakdown of CPU time for the proxy using the *signaling configuration*, with workload generator generating a client arrival rate of 5 clients/second, and a client session duration of 120 seconds, each requesting the v300 video.

Both sets of results show a similar breakdown of CPU time among various operations. We summarize our observations

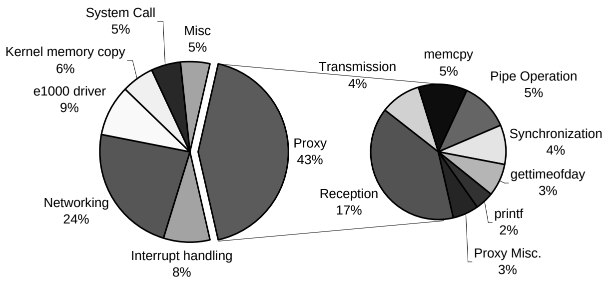


Figure 9: CPU usage breakdown: Data Plane experiment

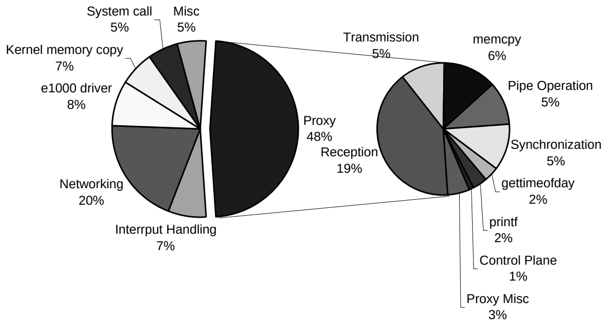


Figure 10: CPU usage breakdown: System experiment

as follows:

- Kernel networking protocol processing takes up a significant percentage of CPU time (23-24%). The top four routines in this category is *udp_v4_lookup_longway*, *sockfd_lookup*, *eth_type_trans*, *skb_recv_datagram*, where the first routine accounts for 38% and 22.59% of the CPU time spent in this category for data plane, and system experiments, respectively. This suggests that future system tuning efforts should focus on these routines.
- Process/thread scheduling overhead, which is shown as part of Misc., is small. For the data configuration experiment, this overhead is only 0.07% of the CPU time; for the system configuration, the overhead is around 0.09%.
- A substantial amount of time is spent in system calls (5%), kernel memory copy (6-7%, including the copies between user space and kernel space).
- Within the proxy application, NRM is CPU intensive. This is due to the functionalities of NRM (receiving packets and introducing frames into pipes), and the complexity associated with handling packet reordering and losses.
- Within the proxy, thread synchronization overhead, which includes thread locking and sleeping operations, is 4-5%.
- The CPU time utilization by the system calls *gettimeofday* and *printf*, which are mainly due to data logging (for performance evaluation purposes) is around 5%.

5 Case Studies

In this section, we describe two case studies that demonstrate how the modular and general design of AMPS allows us to incrementally add new services into the proxy.

5.1 Translation Proxy

We have implemented a prototype proxy that supports reception/forwarding of video streams (MPEG-1) from a video server to clients running RealPlayer. The Windows-based server encodes a live satellite TV-feed to an MPEG-1 stream, and streams the video to the proxy. The proxy translates control signaling messages between RealPlayer and the server. In the data plane, the proxy performs packetization of the MPEG-1 streams that flow into the proxy as a byte-stream to generate the RTP packet flows required by RealPlayer. Through the proxy, multiple RealPlayers could view the live video program smoothly. Please refer to our previous work on translation proxy [41] for details of protocol and format translations. This demonstrates the feasibility of control signaling translation and the generality of the AMPS stream format.

5.2 Interval Caching

Interval caching[17, 10] uses the memory buffer in a video server or proxy to cache video streams in order to decrease the load imposed on the hard disk or the server-proxy network path. An interval-caching proxy caches a moving window of the most recently received content of a video stream. Assuming that a request arrives at time t and that the proxy retrieves the video from the server, the proxy caches the most recent b minutes (referred to as *interval caching length*) of the video, and continuously caches the most recent content as time goes along. Therefore, clients requesting the same video arrived during the time period $[t, t+b]$ can be served from the cache.

Interval caching is easily implemented in AMPS by extending the functionality of the Graph Manager and the pipe class. Recall that a pipe transports a stream of frame pointers from one upstream SGM to multiple downstream SGMs. Each pipe has a schedule indicating the range of frames that should be stored in the pipe. When the first client request for a video arrives, the GM serves the client as usual. Furthermore, the GM records the pipe that serves the client, and sets the interval caching length b by requesting that the pipe retains the last b minutes of frame pointers in its buffer. Our pipe's original implementation used a fixed sized buffer for frame pointers. To prevent buffer overflow for a long interval caching (large b), we extended the pipe so that the buffer size could be initialized to any value. When a client request arrives within the interval that can be serviced by the pipe, the GM forks the pipe and feeds it to the NTM (Network Transmission Module), and instruct the NTM to send the stream to the client. Otherwise the GM serves the request by requesting the stream from the server.

In Section 4.9, we showed that the proxy's CPU usage was 56% with a client arrival rate of 5 arrivals/second and a client session duration of 120 seconds. We conducted an experiment with the same settings, using an 20 second interval cache. In this setting, the proxy needed fewer than 13 streams from the server to satisfy the arriving clients, as opposed to the needed 600 server streams without interval caching. We found that the average CPU usage dropped from 56% to $5.831(\pm 0.222)\%$, since the proxy received only 2% of the network traffic that it received in the no-interval-cache case. In addition, the average client startup latency dropped from the previous 4.16 seconds to 0.43 seconds, since clients served from the interval cache do not experience the server-proxy path delay (including signaling and streaming). Further experiments showed that the proxy could support a client arrival rate of 10 request/second with a CPU usage of $49.43(\pm 0.746)\%$.

6 Conclusions

We have designed and implemented a multimedia streaming 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. The experiments were designed to study the two main components of the proxy - the control and the data plane - separately and jointly.

Our results have shown that the control plane can handle a high arrival rate and a large number of concurrent client sessions efficiently, without incurring long client signaling latency.

We quantified the maximum data throughput that can be supported by the proxy's data plane. We found that the CPU was the proxy's bottleneck after system-tuning to properly size various buffers. We demonstrated that the proxy could sustain up to 288.2 Mbps data throughput, with only a small amount of loss (all of which occurred in the receiving proxy NIC in the server-to-proxy segment).

We showed that the interaction of control and data plane does not incur significant overhead, and helps validate our design decision to clearly separate control and data streaming functionality. The system is able to provide good quality of service (as measured by delay jitter) to the client even under a relative heavy load.

Finally, the case studies demonstrated the generality in the AMPS design allowed us to support signaling translation and data repacketization, and how interval caching could be easily implemented using our configurable data plane building blocks.

References

[1] Linux performance tuning: Web server tuning. http://linuxperf.nl.linux.org/webserving/.

[2] Jussara M. Almeida, Derek L. Eager, and Mary K. Vernon. A hybrid caching strategy for streaming media files. In *MMCN*, 2001.

[3] 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.

[4] G. Banga, P. Drushel, and J.C. Mogul. Better operating system features for faster network servers. In *SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[5] Michael K. Bradshaw, Bing Wang, Subhabrata Sen, Lixin Gao, Jim Kurose, Prashant Shenoy, and Don Towsley. Periodic broadcast and patching services - implementation, measurement, and analysis in an internet streaming video testbed. In *Proceedings of ACM Multimedia System*, 2001.

[6] M. Buddhikot, G. Parulkar, and J. Cox. Design of a large scale multimedia storage server. *Journal of Computer Networks and ISDN Systems*, pages 504—524, Dec 1994.

[7] Y. Chae, K. Guo, M. Buddhikot, S. Suri, and E. Zegura. Silo, rainbow, and caching token: Schemes for scalable, fault tolerant stream caching.

[8] Abhishek Chandra and David Mosberger. Scalability of linux event-dispatch mechanisms. In *USENIX Annual Technical Conference*, June 2001.

[9] Shih-Fu Chang, Alexandros Eleftheriadis, and Dimitris Anastassiou. Developement of columbia's video on demand testbed. *Image Communication Journal:Special Issue on Video on Demand and Inerative TV*, 1996.

[10] Asit Dan and Dinkar Sitaram. Multimedia caching strategies for heterogeneous application and server environments. *Multimedia Tools and Applications*, 4(3):279–312, May 1997.

[11] Derek Eager and Mary Vernon. Dynamic skyscraper broadcasts for video-on-demand. In *Proc.* 4^{th} *Inter. Workshop on Multimedia Information Systems*, September 1998.

[12] Annette C. Feng, Wu-Chun Feng, and Geneva G. Belford. Packet spacing: An enabling mechanism for delivering multimedia content in computational grids. In *The Journal of Supercomputing*, 2002.

[13] Sebastien Godard. Sar. Linux User's Manual.

[14] Mathieu Goutelle. Queues in the linux kernel. http://icfamon.dl.ac.uk/papers/DataTAG-WP2/reports/tast1/20021205-goutelle.pdf, 2002.

[15] Stephane Gruber, Jennifer Rexford, and Andrea Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. *Computer Networks (Amsterdam, Netherlands: 1999)*.

[16] R. Haskin. Tiger shark–a scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[17] Mohan Kamath, Krithi Ramamritham, and Don Towsley. Continuous media sharing in multimedia database systems. In *Proc. of 4th International Conference on Database Systems for Advanced Applications (DASFAA'95)*, April 1995.

[18] Jonathan Lemon, Zhe Wang, Zheng Yang, and Pei Cao. Stream engine: A new kernel interface for high-performance internet streaming servers. In *Web Content Caching and Distribution Workshop (IWCW)*, 2003.

[19] John Levon, Dave Jones, Bob Montgomery, Will Cohen, Graydon Hoare, and Falk Hueffner etc. Oprofile. http://oprofile.sourceforge.net.

[20] Davide Libenzi. Improving (network) i/o performance. http://www.xmailserver.org/linux-patches/nio-improve.html.

[21] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini multimedia storage server. *Multimedia Information Storage and Management , Editor S. M. Chung, Kluwer Academic Publishers*, 1996.

[22] KOM at Darmstadt University of Technology Multimedia Communications Lab. Komproxyd open source project. http://dmz02.kom.e-technik.tu-darmstadt.de/KOMproxyd/.

[23] Makoto Nakamura, Mary Inaba, and Kei Hiraki. Fast ethernet is sometimes faster than gigabit ethernet on lfn - observation of congestion control of tcp streams, 2003.

[24] W.-T. Ooi, B. Smith, S. Mukhopadhyay, H.H. Chan, S. Weiss, and M. Chiu. The dali multimedia software library. In *MMCN*, 1999.

[25] K. Patel and L.A. Rowe. Design and performance of the berkeley continuous media toolkit. In *MMCN*, 1997.

[26] Colin Perkins, Ladan Gharai, Tom Lehman, and Allison Mankin. Experiments with delivery of hdtv over ip networks. In *Proceedings of the 12th International Packet Video Workshop*, 2002.

[27] Niels Provos and Chuck Lever. Scalable network I/O in linux. In *USENIX Technical Conference, FREENIX track*, June 2000.

[28] Reza Rajaie and Jussi Kangasharju. Mocha: a quality adaptive multimedia proxy cache for internet streaming. In *NOSSDAV*, 2001.

[29] Reza Rejaie, Haobo Yu, Mark Handley, and Deborah Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet. In *INFOCOM*, 2000.

[30] Miguel Rio, Tom Kelly, Mathieu Goutelle, Richard Hughes-Jones, Jean-Philippe Martin-Flatin, and Yee-Ting Li. A map of the networking code in linux kernel 2.4.20. http:/datatag.web.cern.ch/datatag/papers/drafts/linux_kernel_map/draft10.pdf.

[31] Sumit Roy, John Ankcorn, and Susie Wee. Architecture of a modular streaming media server for content delivery networks. In *IEEE International Conference on Multimedia and Expo*, 2003.

[32] Jamai Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *USENIX Annual Technical Conference*, November 2001.

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

[34] Subhabrata Sen, Jennifer Rexford, and Don Towsley. Proxy prefix caching for multimedia streams. In *Proc. IEEE INFOCOM*, April 1999.

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

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

[37] Bing Wang, Subhabrata Sen, Micah Adler, and Don Towsley. Optimal proxy cache allocation for efficient streaming media distribution. In *INFOCOM*, 2002.

[38] Yuewei Wang, Zhi-Li Zhang, David Du, and Dongli Su. A network conscious approach to end-to-end video delivery over wide area networks using proxy servers. In *Proc. IEEE INFOCOM*, April 1998.

[39] Robbert van Renesse Wei Tsang Ooi and Brian Smith. Design and implementation of programmable media gateways. In *NOSSDAV*, 2000.

[40] Gary R. Wright and W. Richard Stevens. 16.13 select system call. In *TCP/IP Illustrated, Volume 2, The Implementation*.

[41] Xiaolan Zhang, Don Towsley, and Jack Wileden. Towards interoperable multimedia streaming systems. In *Packet Video Workshop*, 2002.