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

UMass Computer Science Technical Report 2004-08

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 for the two componenets 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 networking protocol and the Network Reception Module inside the proxy to be the most CPU-intensive componenets. We also characterize the end-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.

1 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

[*] This work is done when Yang Guo was a postdoctoral research associate with the Dept. of Computer Science at UMass Amherst

(such as periodic broadcast [7, 15]), proxy prefix caching [43], proxy caching strategies [47, 15, 38, 2, 9], cooperating proxies [37], and streaming CDNs [46].

While there are a number of commercial (e.g., Darwin, RealServer, and Windows Media Server) and experimental streaming servers [3, 11, 30, 45, 7, 27], there is considerably less research on the design and implementation of streaming *proxies*. Existing proxy design and implementation work [20, 40, 24, 4] 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 [34], Dali multimedia software library [32], the open source project GStreamer [21], 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 [49].

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 2 describes the architecture design and implementation of the AMPS platform. Section 3 discusses the experimental setup and performance metrics. Section 4 presents our performance study results. Section 5 demonstrates the generality and flexibility of AMPS design through two case studies. Finally Section 6 concludes the paper.

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

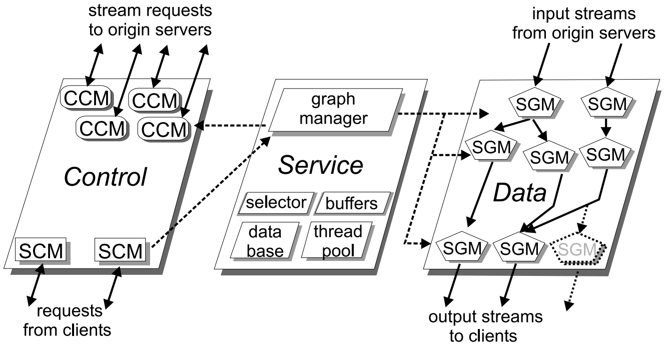


Figure 1: Modules interaction within AMPS Proxy to service a request

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 4.3.1. Our *Selector* is derived from the komproxyd [4], 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 of 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 [42], 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 [49].

2.3.1 Control Plane Implementation

We implemented an SCM and a CCM that speak the RTSP protocol based on the komproxyd RTSP parser [4]. Both modules were implemented as a collection of functions that are executed by threads from a thread pool.

RTSP SCM allows the proxy to act as a RTSP server to the clients. As the current implementations of RTSP protocol of streaming clients often employ a single TCP connection for the control signaling of the total streaming session that lasts for from several minutes to half an hour, the RTSP SCM needs to handle a potentially large number of simultaneous client connections (most of them are idle) efficiently. We employ a *thread-per-request* model as described below: How to refer to the debate between event-driven and thread-based architecture ?

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. By releasing threads between incoming messages, the platform uses fewer threads than a thread-per-client 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.

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 [7].

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 33-ms 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 4.4.2.

The **Network Transmission Module (NTM)** retrieves streams from upstream pipes and transmits the packets to specified network addresses. As previous work [16, 31] has shown that spacing data packets evenly lowers the experienced loss rate, the proxy/server(NTM) ideally should smoothly transmit a stream by spacing output packets evenly at a constant bit rate. However, as [35] has also observed, due to the 10 ms scheduling granularity of Linux, spacing packets through adding sleep interval into the transmission procedure could only achieve packet spacing on the order of tens of millisecond. Another way to introduce packet spacing is through busy waiting. Doing so at the NTM for a large number of streams places a burden on the CPU, and the NTM must share the CPU with other proxy functions. We expect that packet spacing can be better achieved at the IP layer through the Linux Traffic Control mechanism [26].

Our NTM works in rounds. It 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 4.4.3 we describe the tradeoffs in choosing the round length for NTM.

3 Experiment Setup and Performance Metrics

The previous section presents the architecture design and implementation of the AMPS proxy. To study the performance impacts of the architecture design decision and implementation alternatives, explore system tuning issues (such as SGMs round length etc.), and identify system bottleneck within the proxy, we set out to perform a set of experiments in a Gigabit LAN environment. Section 3.1 explains the experiment setup and Section 3.2 describes the performance metrics we reported on for our experiments.

3.1 Experiment 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 Controller) 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

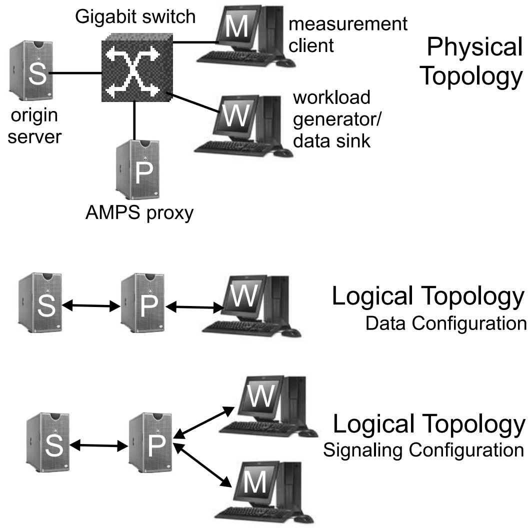


Figure 2: Experimental Configurations

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 seperate NICs in the proxy.

The *data configuration* 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 the ICMP port-unreachable messages.) In the data configuration, the proxy first initiates a certain number of stream to place load on the system, and then periodically adds new streams until the proxy is saturated. 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* 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 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 session arrivals in sequence, and logs the signaling delay and the reception time 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, 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.

video file	frame rate (fps)	bitrate (Kbps)	pkts size (byte)	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

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* [18] 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 will report on the following proxy performance metrics:

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 (Section 4.1). We then present the profiling result of the proxy that provides a system view about the CPU load of various operations within the proxy (Section 4.2). In Section 4.3, we 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. In section 4.4, we study the performance of the data plane, by first comparing the performance of different Network Reception Module design choices, discussing the tuning of the Network Transmission Module, and examining the maximum supportable throughput of the data plane. Finally, in Section 4.5, 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 Issues

During the course of our experiments, we found that substantial amount of system tuning is required to improve the performance of the proxy and to lower the data packet loss rate. Some of the tuning issues have been addressed in the web server and streaming server settings, while the unique workload of a high-end streaming proxy presents new tuning issues.

To increase the number of simultaneous client sessions that the proxy can support, we need 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.

Under the default Linux 2.4.22 settings, we measured significant losses over the server-to-proxy path at relatively low bandwidth utilizations, even if we directly connected the proxy and the server with a cable. Tuning the protocol stack at the sender/receiver side to address packet loss occurring at end-host is relatively new, as other network servers such as Web servers employs TCP for reliable transmission and packet losses are hidden from the application. To identify where packets were being dropped, we studied the Linux networking protocol implementation [39, 19], and found that the default settings were not designed to handle constant high throughput as 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:

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, and the maximum value is 4096 packets (based on the e1000 driver 5.2.20 README file). We enlarged this to 2048 for sending NICs.

RxRing at the receiving NIC: the default value is 256 packets, and the maximum value is 4096 packets. We enlarged this to

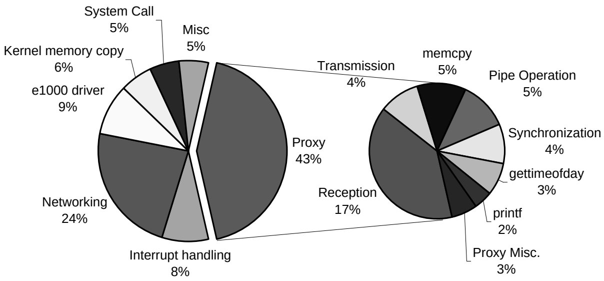


Figure 3: CPU usage breakdown: Data Plane experiment

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. Previous work [35] also pointed out the need for enlarging the socket buffer for receiving high data rate video.

After these changes, we were able to increase the data throughput between the server and the proxy until the proxy's CPU was saturated, with a small amount of losses occurring at the proxy's receiving NIC (the *rx_ovr* error reported by *netstat*).

> **Lesson: To handle a high volume of network traffic, it is necessary to tune the buffer sizes used at various levels of the network protocol stack for both the sending and receiving sides. The buffer needs to be large enough to absorb reasonable bursts of traffic, as well as delays in processing data.**

4.2 Proxy Profiling Results Analysis

In our testing environment, CPU has been identified as the system bottleneck. To understand the CPU usage within the proxy, we employ Oprofile [28] to profile the proxy. In our profiling setting, OProfile uses the real-time clock interrput to collect samples of the PC (Program Counter) value, so that an percentage of CPU spent in various routines can be estimated. We ran an experiment using the data configuration with a fixed number (600) of the v300 streams, and an experiment using signaling configuration, with the workload generator generating a client arrival rate of 5 clients/second, and a client session duration of 120 seconds, each requesting the v300 video. In this signaling configuration experiment, the average number of sessions in the proxy is 600 and the average data plane throughput is 180 Mbps. The CPU idle time for the data configuration and signaling configuration experiment is 45% and 34% respectively.

We analyzed the profiling result, categorized the system/user routines, and ploted the breakdown of CPU time for both experiments in Figure 3, and Figure 4. In both figures, the left pie shows the breakdown of CPU time among various kernel functionalities and the proxy, and the right pie further represents the breakdown of CPU time among proxy routines. Both sets of results show a similar breakdown of CPU time among various operations. Below we focus on the signaling configuration experiment profiling result.

Compared to the profiling result on a streaming server reported in [27] where disk access is the top routines that consumes

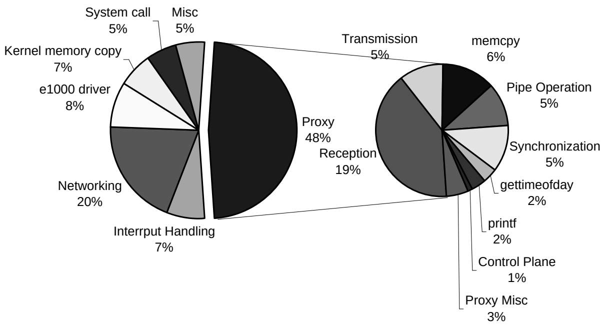


Figure 4: CPU usage breakdown: System experiment

CPU, the streaming proxy shows a different system bottleneck. We observed that kernel network protocol processing takes up a significant percentage of CPU time (20%). As the proxy handles the reception and transmission of a large amount of network data packets, the network protocol processing (both the sending and receiving path) 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 with 128 entries that map destination port number to the kernel socket structure. 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.

Similar to the streaming server in [27], 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 space and kernel space). This demonstrates the overhead of user-kernel context switches, and the benefit of adopting schemes that eliminate data copying [13, 33].

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

The profiling result also quantifies the overhead of using pipe to pass streams among SGMs. Pipe operations, which include querying the schedule of a pipe, poping/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 design of AMPS avoids the expensive memory copy operations by passing streams among SGMs by frame pointers (instead of memory copy). 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 just before being sent out. 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 wouldn't gain a huge amount of performance improvement.

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

We notice that the nature of proxy workload makes its performance greatly dependent on the OS network stack implemenation. For example, 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 [41, 39] introduced to Linux since version 2.4.19. This model allows the CPU to 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.

4.3 Control Plane Performance

The profiling result in Section 4.2 shows that control plane is not the bottleneck for the basic receive-and-forward proxy. One can expect that with the deployment of services such as prefix caching and periodic broadcast [23, 17, 15, 14, 44, 8, 22] that alleviate stress on data streaming components, control plane will be more stressed and is likely to become system bottleneck.

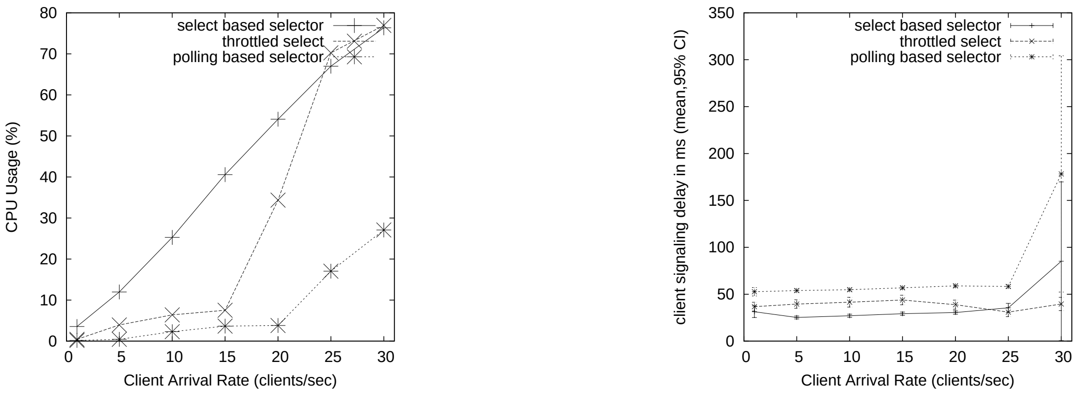
In this section, we explored two design/implementation issues that impact the control plane performance: selector implementation (Section 4.3.1), and threading model (Section 4.3.2).

4.3.1 Impact of Different Event-Dispatching Implementations

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, a set of 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 varying client arrival rate, and a client session duration of 120 seconds. We plot proxy CPU usage and client signaling delay using several selector implementations in Figure 5.

We first consider using the *select* system call to monitor the interest set. In this approach, the selector thread runs in rounds. In each round, the selector thread first initializes the interest sets, calls *select* with the interest sets, and upon return from the *select* call checks for ready file descriptors and calls the associated callback functions. After serving all ready file descriptors, the thread proceeds to the next round. As Figure 5.(a) shows, the incurred CPU usage increases linearly with the client arrival rate. The implementation of the *select* call in Linux conforms to the 4.4BSD implementation, for which [48]



(a) Proxy CPU usage

(b) Client signaling delay

Figure 5: Performance of different selector implementation

has provided a detailed discussion. The improvement for the *select* implementation proposed in [6] hasn't been incorporated.

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. Under high client arrival rates, this approach decreases the frequency with which *select* is called, thus allowing more file descriptors to become readable in each round. Therefore, the cost of initializing interest sets and performing the *select* call is amortized over the multiple ready file descriptors being selected. Figure 5 shows that this approach provides significant CPU saving. As the client arrival rate increases, the CPU usage increases slowly when the client arrival rate is smaller than 15 clients/sec, and increases much faster and converges to select based selector when the client arrival rate is 25 clients/sec. Figure 5.(b) shows that the client signaling delay under the throttled selector is only slightly higher than the *select* based selector.

Our final experiment was to implement the selector using a simple polling mechanism(not to be confused with the *poll* system call). In this approach, the selector directly checks each of the file descriptors to see if it is readable (using the *recv* system call with *MSG_PEEK* option to peek at one byte of data from the socket), and calls the associated callback function if there is data to be read. The selector sleeps for 15 ms between consective pollings. This approach further reduces CPU usage. For example, at the client arrival rate of 20 clients/sec, the polling based selector incurs a CPU usage of 3.81%, whereas *select* based selector and throttled selector incur CPU usages of 54.07% and 34.37% respectively. Compared to the *select* based selector, the increase in client signaling delay under the polling based selector is bounded by 30 ms. The polling based selector avoids the overhead in specifying interest sets (which results in kernel-user space memory copy) and scanning the interest sets, and therefore, achieves further reduction in CPU usage. We found that the saving in CPU utilization justifies the increase in delay of 30 ms. 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 [6, 5, 36, 10]. 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.

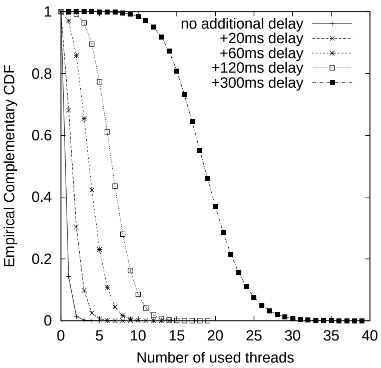


Figure 6: Threads Usage under various server-proxy signaling delay

Two new event-notification mechanisms recently introduced into Linux are the */dev/poll* [36] and *epoll* [29](which partially implemented the scheme proposed in [5]). We didn't make use of these system 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.**

4.3.2 Impact of control plane threading model

As described in Section 2.3.1, we employ a thread-per-request model to implement the 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 impact 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 dominates the holding time of the thread and therefore the total number of threads used in the control plane. In our testing environment, the server-proxy signaling delay is less than 10ms. To simulate a longer server-proxy signaling delay, we instrumented the proxy so that an 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 6 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 increases 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 dynamically changing, we can adapt the thread pool size based on current thread pool usage and system load.

> **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 Data Plane Performance Measurements

In this section, we investigate design and tuning issues concerning the two data plane modules, the NRM (Section 4.4.2) and the NTM (Section 4.4.3), and report the maximum data plane throughput supported (Section 4.4.4). We used the data configuration (as shown in Figure 2) for all the data plane experiments.

4.4.1 Fluctuations in CPU Usage for Data Plane Experiments

We used the *data configuration* (as shown in Figure 2) to study data plane performance. While the load increases linearly in this configuration, we observed certain unexplained CPU usage fluctuation as shown in Figure 7, under a load of 150 to 300 streams using *select*, and under a load of 380 to 420 streams using polling. Figures 8 and 9 reveal similar phenomena. The fluctuations reappear at different times during repeated trials of the same experiments. 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 the data configuration experiments. The profiling information was saved for each 100 second epoch of the experiment. We then compared the profiling results during the peak of the fluctuations and 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 interrupt rate, 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 on the trend revealed by the plots in the remainder of this paper.

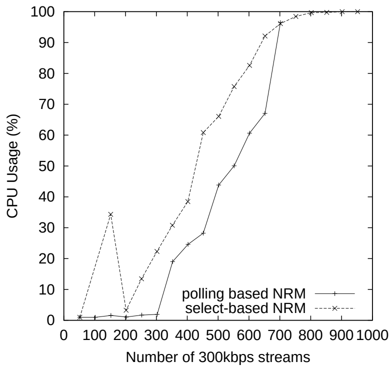


Figure 7: Polling vs select based Reception Module

4.4.2 Performance Optimization for 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. Profiling result in Section 4.2 has shown that the NRM is the most CPU intensive component within the proxy. We discussed the optimization we applied to the NRM in this section.

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 [5, 10], 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 7 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.

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.

4.4.3 Network Transmission Module

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

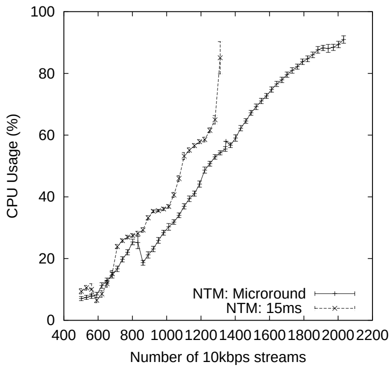


Figure 8: CPU saving of NTM Microround scheme

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.

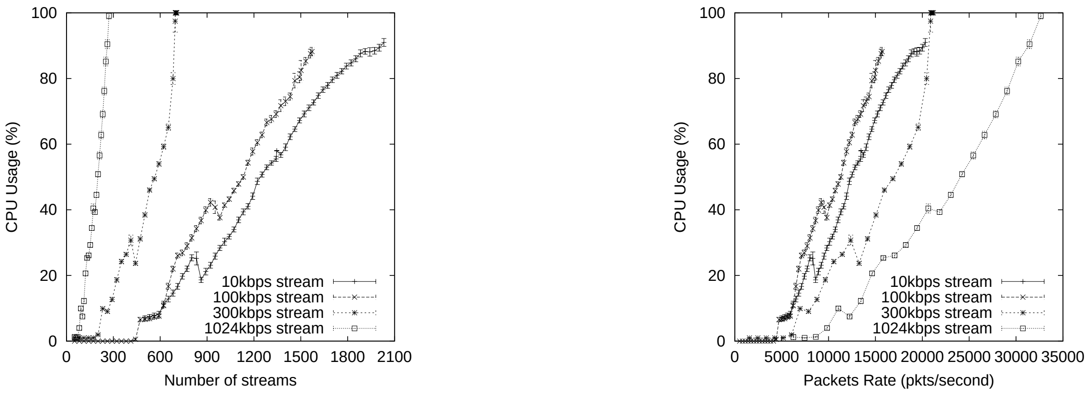
Due to the 10 ms scheduling granularity of Linux 2.4.22, using a round length of less than 15 ms results in consistent oversleeping by the NTM thread. Given this constraint, we implemented two different solutions. In one implementation, we set the round length of the NTM to be 15 ms (the smallest sustainable 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 8. 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.

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

4.4.4 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. Table 2 summarizes the maximum number



(a) Proxy CPU usage vs number of streams

(b) Proxy CPU usage vs packets rate

Figure 9: Scalability of Data Plane

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

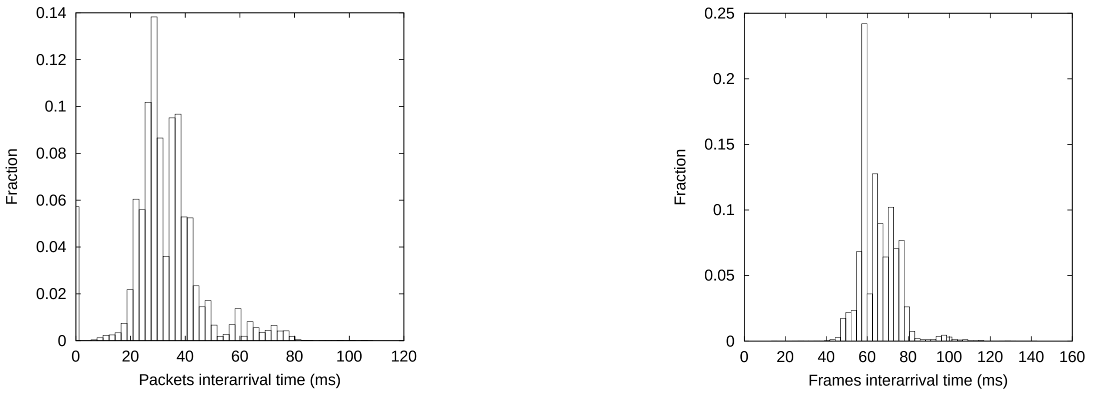
Table 2: Maximum Data Plane Throughput

of video streams and data throughput that the proxy can support 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.

We further characterize the proxy CPU usage under different data plane workload. For different videos, Figure 9(a) plots the relationship between the CPU utilization and the number of video streams that the proxy receives and forwards, and Figure 9(b) plots the relationship between the CPU utilization and the packets reception/forwarding rate.

The results show that while proxy CPU usage increases (as expected) with an increasing data rate, the number of video streams being handled also influences utilization. For the same overall data rate, the configurations with a smaller number of higher bitrate videos had lower CPU utilization than configurations with a larger number of lower bitrate videos.

The work reported in [31] 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



(a) histogram packet interarrival time

(b) frame interarrival time

Figure 10: Client Reception Quality with client arrival rate of 5

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.

4.5 Overall System Performance

For overall system performance, we analyze the video delivering quality as observed by measuring clients.

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 only the first 120 seconds of the video. We found the proxy could sustain client arrival rates of up to 5 clients/sec which results 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 10 plots the histogram of the client packet- and frame-interarrival time. The mean of the frame interarrival 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 initial buffering of 52 millisecond at the client is enough to ensure smooth playback (no buffer underflow).

The average client startup latency is 4.18 seconds. Most of this delay can be attributed to the manner in which the proxy and the server serves a new stream request. 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 MLM(Memory Loader Module). When the proxy receives response from the server, it commit itself to start forward the stream to the client in two second. This is used to smooth the jitter between the proxy and the server.

For all experiments, the measuring client observed no data packet 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 [35], where they conjectured that a race condition in Linux kernel resulted in the reordering.

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 [49] 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[25, 12] 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 4.5, we showed that the proxy could sustain 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 56.88% to 5.83%, 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 clients/second with a CPU usage of 49.43%.

6 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.

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.

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

References

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

[2] 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.

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

[5] 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.

[6] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[7] 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.

[8] Steven Carter and Darrell Long. Improving video-on-demand server efficiency through stream tapping. In *Proc. International Conference on Computer Communications and Networks*, 1997.

[9] 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.

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

[11] 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.

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

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

[14] D. Eager, M. Ferris, and M. Vernon. Optimized regional caching for on-demand data delivery. In *Proc. Multimedia Computing and Networking*, January 1999.

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

[16] 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.

[17] Lixin Gao, Jim Kurose, and Don Towsley. Efficient schemes for broadcasting popular videos. In *Proc. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

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

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

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

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

[22] Kien Hua, Ying Cai, and Simon Sheu. Patching: A multicast technique for true video-on-demand services. In *Proc. ACM Multimedia*, September 1998.

[23] Kien Hua and Simon Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proc. ACM SIGCOMM*, September 1997.

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

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

[26] Alexey Kuznetsov. Linux traffic control. http://www.sparre.dk/pub/linux/tc/.

[27] 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.

[28] J. Levon and et al. OProfile. http://oprofile.sourceforge.net.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

[39] 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.

[40] 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.

[41] Jamai Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *5th Annual Linux Showcase & Conference*, November 2001.

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

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

[44] Subhabrata Sen, Lixin Gao, Jennifer Rexford, and Don Towsley. Optimal patching schemes for efficient multimedia streaming. Technical Report 99-22, Department of Computer Science, University of Massachusetts Amherst, 1999.

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

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

[47] 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.

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

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