TECHNIQUES FOR IMPROVING MULTIMEDIA COMMUNICATION OVER WIDE AREA NETWORKS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Soamdev Acharya

January 1999

TECHNIQUES FOR IMPROVING MULTIMEDIA COMMUNICATION OVER WIDE AREA NETWORKS

Soam Acharya, Ph.D.

Cornell University 1999

Despite widespread interest and technical progress, significant barriers exist for video playback over the Internet. These obstacles include network unreliability, client heterogeneity, and server bottlenecks. Although various playback systems have been proposed, none address all the issues satisfactorily. This thesis proposes and investigates MiddleMan, an alternate approach.

MiddleMan is a collection of cooperating caching proxies running in a local area network (LAN). Such a configuration offers several advantages. By caching videos relatively close to the clients, MiddleMan reduces overall startup delays and the possibility of adverse Internet conditions disrupting video playback. Additionally, MiddleMan dramatically reduces server load by intercepting a large number of server accesses and can be easily extended to provide other services.

Several issues must be addressed before MiddleMan can be built and deployed. The first problem involves determining the intrinsic properties of video files on the web and how they are accessed over the Internet. Such an understanding is useful in order to effectively detail the architecture of MiddleMan. Hence, I conducted two studies: one to characterize videos on the web and another that analyzes how users access videos. I then used these results to derive the architecture of MiddleMan.

The second hindrance to building MiddleMan involves evaluating and refining the design. Hence, I developed a simulation environment for MiddleMan to test various configurations and caching algorithms. The final design achieves both high cache hit rates and excellent proxy load distribution.

Finally, MiddleMan supports client heterogeneity by converting video to an intermediate format that allows the system to better adjust to client loads. Thus, techniques for fast conversion of video must be developed and integrated into MiddleMan. Hence, I developed a compressed domain transcoder that converts MPEG to JPEG. The transcoder is about 1.5 to 3 times faster than its spatial domain counterpart.

BIOGRAPHICAL SKETCH

Soam Acharya was born in London, U.K., in the same year as the height of the “flower power” movement, the release of the “White Album,” and a year before the first landing on the moon. Coincidence? Pure chance? He disclaims responsibility for any of these events. He was then transported to Calcutta, India, but left for Iran just before the Marxists came to power in West Bengal. After two years in Iran, he departed for Kuwait, six months before the Iranian Revolution. He spent the majority of his adolescence in Kuwait, graduating from the New English School in 1986 and joining the undergraduate program at the University of Texas, Austin. By the time the Iraqis ran over Kuwait, he was living in Dallas, Texas, working at the Superconducting Super Collider Laboratories. While taking on part-time jobs to keep body and soul together, both at Lawrence Berkeley Laboratories and at Cornell University, he completed an M.S.E.E. at Cornell in 1993. He entered the Ph.D. program full-time that same year. He currently has no plans to leave the U.S.A.

To the force that through the green fuse drives the flower.

ACKNOWLEDGEMENTS

At the end of it all, I find myself at a loss for words. How does one adequately distill and compress all these years of color, heartburn, frostbite, loves, losses, music, light, sweat, and vision into a few choice words to thank those responsible? Well, life goes on and, prior to resuming mine again, here follows my attempt at highlighting the contributions of those that sustained and taught me while I was in suspended animation.

My advisor, Brian Smith, instilled in me the desire for excellence, monitoring my growth, shaping my graduate career and giving it focus. He was my guru in the true sense of the word. Thanks also to Robert Thomas and Geri Gay for their valuable guidance.

My parents deserve canonization for my pain was theirs. They gave me discipline, conscience, heritage, and values and without those, I would be nothing. Joydev, my brother, loved me like one and proved that, in the end, “blood is thicker than the mud,” as Sly and the Family Stone would say. During times of great difficulty, I derived inspiration from the struggles of my cousins, Ranadev, Basudev, Sudeshna, and Pushan, and I have no doubt of their eventual success. My uncles, Satyendra Prasanna, Sourindra Prasanna, and Shishir Chandra, have been role models and I can but aspire to their achievements. Although my grandfathers did not survive to see this day, I know they live on in me and I also know that somewhere in the sky, they are happy for I have made them proud.

Manish Kothari has raised the bar by which I judge all friendships. Mark Barrera has taught me patience and “given” me his copy of “Stevie Wonder’s

Original Musiquarium." What more can a man ask? Mark Riccio has converted me to the virtues of bread, home theater systems, tennis and lengthy expectorations and for those I am forever grateful. Anindya Basu is the one person in the world that shares a sense of humor similar to mine. Someday, I shall have the privilege of inviting him to the premiere of "Curry In Jersey." P. Subrahmanya has been my partner in crime for our creative endeavors. Together, we have made people laugh and listen and our efforts have given me an education as valuable as my formal one. Sayandev Mukherjee, has been a valuable source of strength and Dove bars. Paul Chang, Tibor Janosi, Nicole Dreyfus, Koeunyi Bae, and Annie Qahir have provided much counseling over the years. Rosa Garcia was kindness itself. My philosophical discussions with Felix Ejeckam led to the brink of world peace but were invariably interrupted by the bill arriving. Srikant Jayaraman owes me a fruit cake but other than that, has provided impeccable support. Sugata Mukhopadhay has patiently listened to my rantings and ravings and sometimes, he has even agreed. We'll breach the 250 pound barrier together and that's a promise. Mike Novak, George Mathew and Rohit Mehrotra befriended me when my "coolness quotient" used to be high and stayed on famously. Anne Kayani became my first leading lady and, someday, I'll do her trust justice. Thanks also to Marilyn Rivchin for legitimizing my multimedia aspirations. Finally, Lady D gave me my own "Philadelphia Story" and I cherish it.

My lab mates Chi-Chao Chang, Greg Czajkowski, Chris Hawblitzel, Weitsang Ooi, and Athanasios Kyparlis have endured me constantly and I am grateful for their tolerance. John Belina, despite his schedule, has always had time for my ramblings. Last but not least, Cindy Robinson has been an oasis amidst the sandstorm. Her insight and wisdom kept me sane during some of the most stressful times at Cornell.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The first video transmission systems on the Internet were deployed in the early 1990s. These systems were primarily intended for conferencing and live broadcast applications and included the vic/vat system from UC Berkeley [34] and CUSeeMe from Cornell University [78]. It was not until the emergence of the World Wide Web (1993-1994) with its systematic approach to serving and viewing non-text documents that playback of stored video content over the Internet (video on the web, or VOW) emerged as a nascent application to most users. However, Internet video playback was marred by lengthy download times and jerky playback on the client. To address these drawbacks, companies such as VXTreme [76], Progressive Networks [79] and VDOLive [77] utilized experimental systems to develop products that "streamed" videos to users on demand. By 1998, video on the web had seen significant commercial interest. For example,

1. news web sites such as ESPN and CNN have dedicated sections devoted to audio and video content. Other sites supplement their HTML content with audio and/or video clips. The Internet broadcast of President Clinton’s grand jury testimony, for example, attracted hundreds of thousands of viewers on the day of its availability in 1998 [82].
2. a growing number of “micro-broadcasting” stations distribute audio/video material exclusively via the Internet. These include Free Speech Internet Television [85], a website devoted to “alternative” media, Jagged Internet-

works [80], a company founded to cybercast cricket matches, Streamland [83], a forum for viewing music videos, and broadcast.com, which offers a large selection of programming including sports, talk and music radio and movies to an average of 460,000 unique users daily [84].

3. educational institutions such as Stanford [88], Cornell [87] and Georgia Tech [86] are making video recordings of course lectures and slides available online.

Despite the widespread interest and technical progress, significant barriers still exist for video playback over the Internet. These obstacles include network unreliability, client heterogeneity, and server bottlenecks. Some hindrances will vanish with time, others will persist. I outline the current issues involved in each problem in the next three sections. In the fourth section, I speculate which of these drawbacks are likely to be removed by technological progress and which are not.

Network Unreliability

From the user's perspective, the biggest drawback with the traditional download-and-play model is download time. The large size of video files, coupled with the unreliability of the Internet, lead to seemingly interminable wait times. Hence, a number of streaming protocols/systems [76,77,79] have been proposed that attempt to ameliorate the situation by:

- reducing startup delays: clients can play as soon as a small amount has been received from the server. Thus, the user does not have to wait to download the entire file. Note that the streaming server is often different from the web server which received the original video playback request.

- selectively degrading the data: a streaming system copes with varying network conditions by either dropping video frames, scaling video quality, or both.

Some streaming systems use conventionally encoded video files (QuickTime [73], AVI [72], MPEG [25]) whereas others have developed custom encoders to increase flexibility in transmitting and displaying video data. Streaming systems that use their own custom encoders provide better performance, but video quality is still poor at low bandwidths.

Streaming, however, cannot surmount the lack of Internet QOS guarantees. High network latencies, for instance, can still disrupt video playback. Resrource reservation protocols such as RSVP [64], promise to provide QOS over the Internet. Nevertheless, most client connections to the Internet have insufficient bandwidth for streaming of conventionally encoded video files. My survey of video data on the web in chapter 3 shows that the Internet needs to provide sustained bandwidths of approximately 1 Mbps in order to accomplish this goal, an order of magnitude higher than what is commonly available today.

Client Heterogeneity

Differences in client hardware, software, or network connectivity can cause a video to play well on one machine, but poorly on another. Factors such as CPU speeds, bus rates, disk speeds, availability of hardware video decoders, competing processes, and operating system ineffectiveness all affect software playback performance. Another factor is how clients are connected to the Internet. Some access it via 56K modem lines to ISPs whereas others are directly connected via high speed local area networks. Hence, two otherwise identically configured machines with different network connectivities are likely to differ vastly in playback performance of streaming video. It is difficult to design an architecture that can cope with all the possible variations without compromising video quality.

Additional client-side complexity arises in the form of multiple video formats. Numerous video compression and transmission schemes have been proposed over the years. For example, the latest AVI FAQ [72] lists eighteen video codecs that are available for AVI/NetShow alone. Hence, viewing videos can turn into an arduous exercise in installing and updating codecs.

Server Bottlenecks

The growth of VOW also places tremendous load on web and streaming servers. Web servers must serve video documents in addition to images, text documents, and other binaries. Videos tend to be large and hence consume server-side machine and network resources that could service non-video requests. Streaming protocols transfer load to a separate (streaming) server, but that only has the effect of making the streaming server the bottleneck for video access. Sites with large media collections can attempt to solve this issue either by placing multiple load-balanced streaming servers within their own network or by outsourcing video accesses to other sites. The former approach shifts the pressure point to the network gateway of the media site, whereas the latter approach transfers the bottleneck to the other outsourced sites. Both approaches are still susceptible to unreliable network behavior which can severely impair video data transmission. Additionally, both are expensive solutions and thus, not possible for most sites.

Effect of Technological Progress

The next generation Internet promises to provide more reliable service guarantees, hence the problems due to network unreliability may not be a factor in the future. Furthermore, consolidation in the multimedia industry will result in a few widely used video transmission and playback standards. However, server bottleneck related problems are likely to persist despite improvements in the network fabric. Better connectivity further exacerbates server load as more clients are able to access the same server at higher bandwidths. Similarly, despite hardware and software improvements, competing processes on a client can still affect its video playback performance. In the next section, I propose a solution that alleviates the server bottleneck problem and addresses the other issues.

1.1 Solution To VOW Viewing Problem

In this thesis, I detail an approach to solving the VOW viewing problem. The key observation that drives this method is that requests to a video server tend to exhibit locality of reference. Some videos are much more popular than others. Hence, it is possible to exploit *caching techniques* that reduce redundant video accesses to the server.

Figure 1.1 outlines how caching can be exploited on a system comprised of a remotely located web server and two machines with a high degree of connectivity. Instead of contacting the server directly for a video, the client browser, as shown in step 1, contacts a proxy server running locally on the same machine. The proxy server P1 checks to see if it has stored this title locally. If it

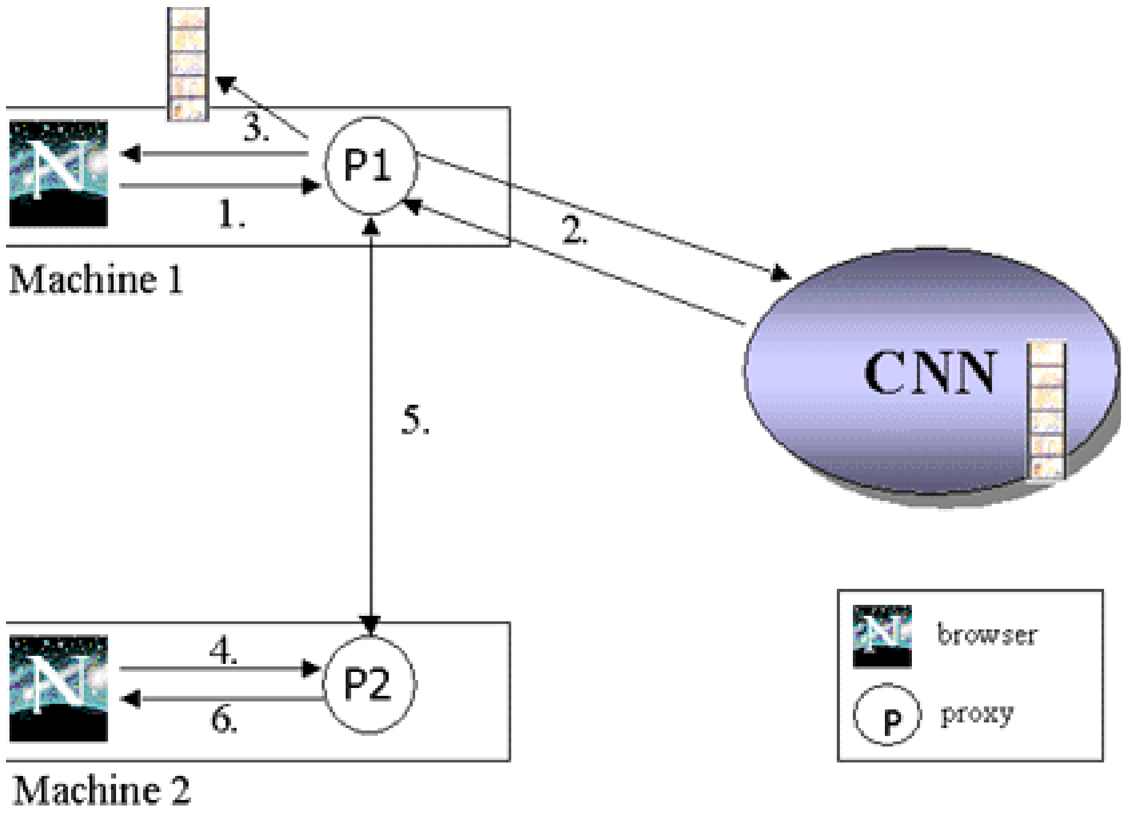


Figure 1.1: Basic Idea For Caching Architecture

does not, it contacts the server directly (step 2) and starts receiving the video. P1 stores a copy of the video locally while simultaneously forwarding another copy to the client browser (step 3). Suppose a user on machine 2 requests the same video. P2 checks to see if a copy of the video is available locally (step 4). Since P1 has a copy, P2 contacts P1 (step 5), accesses the video and forwards it to the local browser (step 6), thus bypassing downloading the video from the original web server.

A *coordinator* process keeps track of the files hosted by each proxy and redirect requests accordingly (figure 1.2). In the previous example, it is the coordinator that directs P2 to P1. The coordinator also uses the proxies to manage the copied video files stored at each machine. If there is no free space left in the

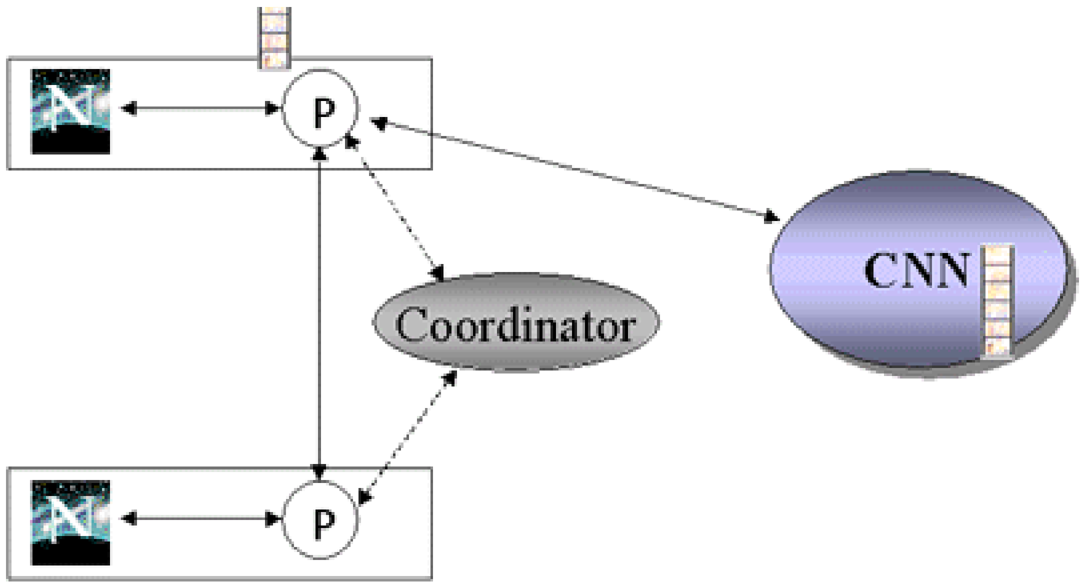


Figure 1.2: Role of the Coordinator

system, it is the coordinator that decides which files to eliminate in order to make room.

To summarize, the architecture I propose, called MiddleMan, consists of a collection of caching video proxy servers that are organized by coordinators. Three technological advances, high speed local networks, high processor speeds, and cheap disk space make MiddleMan feasible. High speed of networks enable the MiddleMan system to quickly service user requests. High CPU speeds allow fast intermediate processing of videos by proxies. Finally, cheap disks provide the large storage space required to cache videos. For instance, consider a campus network consisting of 100 machines, each with 4 GB of disk storage. If each machine allocated 100 MB of space towards the cache (2.5% of total disk space), an aggregate global cache size of 10 GB of space can be achieved.

MiddleMan offers a potentially rich set of benefits in dealing with VOW problems. As can be seen in chapter 6, it achieves high cache hit rates with a relatively small cache size. Hence, by caching videos relatively close to the clients and ensuring a large number of video requests are satisfied locally, it reduces overall startup delays and the possibility of adverse Internet conditions disrupting video playback. From the point of view of the server, MiddleMan dramatically reduces load by intercepting a large number of server accesses. Hence, the net effect of MiddleMan is to greatly increase the *effective bandwidth* of the entire video delivery system allowing more clients to be serviced at any given time.

By incorporating video processing into proxies, MiddleMan offers a mechanism to cope with heterogeneous clients and their variable loads. In particular, proxies can convert incoming video to a format more flexible for display by host machines.

Because proxies are colocated with their clients, MiddleMan can adjust more quickly to both fluctuations in the client load and the local network bandwidth.

1.2 Technical Contributions

Several issues must be addressed before MiddleMan can be built and deployed.

1. *Determining the intrinsic properties of video files on the web and how they are accessed over the Internet.* Not much is known about either, but such an understanding is useful in order to effectively detail the architecture and protocols of MiddleMan.

2. *Evaluating and refining the design.* This process includes determining the effectiveness of various cache replacement and load balancing policies across proxies.
3. *Supporting client heterogenieity.* MiddleMan supports client heterogeneity by converting video to some intermediate format that allows the system to better adjust to client loads. Hence, techniques for fast conversion of video must be developed.

To study the properties of the videos on the web, I devised and executed an experiment. In this experiment, I downloaded and analyzed over 57000 AVI, QuickTime and MPEG files stored on the Web - approximately 100 Gigabytes of data. Among the more interesting discoveries, I verified the conjecture that current Internet bandwidth is at least an order of magnitude too slow to support streaming playback of most of these videos.

To investigate how users access video data, I inspected log file records from the mStar [45] experiment at Lulea University in Sweden. mStar is a hardware/software infrastructure developed by the Center for Distance-spanning Technology at Lulea University for facilitating distance learning and creating a virtual student community. This analysis yielded several insights into video browsing behavior including:

- users do not necessarily view a video all the way through. Only about 55% of all requests went to completion.
- video access patterns exhibit high temporal locality.
- the access pattern of a video is related to its content category. Movie popularity tends to be evenly distributed with time, whereas the popularity of course materials varies greatly with time.

The insights from these two analyses influenced the design of MiddleMan. Armed with the results from these analyses, I developed a simulation environment in order to test the initial architecture. The main design flaw revealed by the simulations was that of poor load balancing between the proxes. After further revisions in the architecture, the final design achieved both high cache hit rates and excellent proxy load distribution.

To address problem 3, I examined the issues involving the fast software conversion of DCT based, inter-frame compressed video formats to motion JPEG. I chose this class of formats because, in addition to the popularity of MPEG-1, technologies such as H.261 [59] and MPEG-4 [95] utilize similar compression concepts are gaining widespread acceptance on the web. *Transcoding* these formats to motion JPEG offers several advantages:

- constraining the proxy-client exchanges to motion JPEG allows for more leeway in client load control. A proxy can drop arbitrary frames if necessary in response to sudden client loads or local network bottlenecks. This response is not usually possible in formats that do not allow random frame access.
- software JPEG decoders are easily available. Most, if not all, hardware platforms have decoders available as part of a standard suite of decoders.

The remainder of this thesis is organized as follows. In chapters 2 and 3, I describe the study to characterize videos on the web and the results of the video trace analysis respectively. I present the initial design and protocols of MiddleMan in chapter 4. In chapter 5, I detail the simulation techniques used to evaluate MiddleMan and use the simulation results to investigate the caching and load balancing properties of MiddleMan and to refine the design. In chapter

6, I investigate techniques for fast MPEG to JPEG conversion. I present my conclusions in chapter 7.

Chapter 2

Characterizing Videos On The Web

2.1 Introduction

Understanding the characteristics of video currently transmitted on the Internet is an essential first step prior to proposing improvements in communication schemes for continuous media data. To effectively structure such a study, I turned to analogous efforts by system researchers designing file systems. It is instructive to see how they approached the issue: for example, in 1985, researchers at the University of California at Berkeley published a study of the UNIX 4.2 BSD file system [43]. This analysis provided a number of insights regarding file sizes, lifetimes, and access patterns and was highly influential in the design of several file systems. In 1991, Berkeley researchers released a follow-up study on Sprite [8], which verified the assertions made in [43] and made further measurements. This study was also influential in the design of subsequent distributed file systems. Following the example set by the two file system studies, this chapter presents the results of a study to categorize video data currently stored on the Internet.

To answer these questions, I wrote a Web analysis system that downloaded and analyzed every video it was able to find on the Web. My goal was to answer the following questions:

- What are the basic properties (size, frame rate, picture dimensions, duration, and average bitrate) of video files?
- How are these characteristics changing?
- How do different compression technologies compare with each other in practice?
- How does network bandwidth affect the waiting times for movie file download?
- Are video files transitory? How long do they last on the World Wide Web?

The answers to these questions are based on 57076 (subsequently whittled down to 47457) video data files downloaded from about 9336 WWW servers in April and May 1997. My findings included:

1. Movie sizes range from hundreds of kilobytes to several megabytes: 1.1MByte is a good “rule of thumb.”
2. Most movies are brief: 90% last 45 seconds or less.
3. Users adhere to "standard" small or medium picture dimensions (such as 160x120 or 320x240) when creating videos with audio content, but not when creating videos without audio.
4. The number of movies coming on-line is increasing exponentially.
5. MPEG [25] files compress better than QuickTime [73] or AVI [72] due to their lower bits/pixel values (mean of 0.73 as opposed to 2.16 or 2.51). MPEG files also have smaller durations and higher frame rates.
6. QuickTime and AVI files have analogous distributions for frame rates, duration, size and waiting time. Bits/pixel comparisons show QuickTime to compress slightly better.

7. 28.8K, 56K and 128K bandwidths are practically useless for real-time display of video data. For example, 56Kbits/sec allows about 1% of all movie files to be downloaded. About half the movies can be displayed with 700 Kbits/sec of bandwidth, 80% with 1.5 Mbits/sec and 90% with 2 Mbits/sec.
8. Movies follow the write-once/read-many principle of availability. 80% of the movies initially analyzed were still present on the web about 4-6 months later.

The rest of this chapter describes this experiment and its consequences in detail. Section 2.2 presents my method for locating, collecting, and processing the video data. Section 2.3 presents the results of thirteen analyses I performed on the collected data set, elaborating the results outlined above. I outline related work in section 2.4 and summarize my conclusions and give directions for future work in section 2.5.

2.2 Video Data Collection Process

For the study to be representative of conditions on the Web, I had to locate and analyze a large number of video files. I divided this task into three steps. The first step, the *hunting phase*, was to obtain a list of links to *video documents*. A video document is an HTML document that contains at least one link to a video clip. The second step, the *gathering phase*, consisted of extracting the video links from each video document, fetching the specified clip, analyzing it, and recording the results. I found a small amount of the data to be suspicious, so the final step, the *sifting phase*, eliminated this suspicious data. The next three subsections describe these phases in detail.

2.2.1 Step 1: The Hunting Phase

To obtain a list of video documents, I wrote a Tcl script [71] to coerce the Alta Vista search engine [70] to return a list of potential video documents. A potential video document is one that contains a link to an MPEG, QuickTime, or AVI file. File extensions of the URLs embedded in the video document were used to distinguish between video and non-video links. Since file extensions are unique in the WWW framework, this approach was sufficient to identify the embedded video links. The search was limited to AVI, MPEG and QuickTime files since these are the most established video technologies. Alta Vista is capable of ordering query results by the date of last modification and this facility was useful in categorizing the retrieved video document links on a month by month basis, from January 1995 to March 1997. The process yielded about 44000 links, of which 22600 were valid.

2.2.2 Step 2: The Gathering Phase

Armed with the list of potential video documents, I wrote a system to download the video documents, and contained video files. The video link processing system, shown in figure 1, consists of a link distributor and gatherer (LDG) process and a set of link processor (LP) processes. The LDG is responsible for assigning video documents to LPs and collecting and storing the summary statistical data calculated by the LPs. Upon obtaining a video document URL, the LP fetches the document, parses it to extract any links to movie clips, downloads the clips, runs a video analysis program, and sends summary statistics to the LDG. These statistics include the basic movie properties (frame rate, clip size in bytes, etc.) as well as properties of the video document (modification time, size in bytes).

For example, suppose the video document http://www.eg.com/movie.html contains a link to http://www.hoho.com/my.mov. Figure 2.1 illustrates the steps the video link processing system would take in processing this video document:

1. LP1 requests a new video document link from the LDG. It receives the URL http://www.eg.com/movie.html
2. LP1 contacts www.eg.com and fetches movie.html
3. LP1 parses the contents of movie.html and extracts the link http://www.hoho.com/my.mov.
4. LP1 contacts www.hoho.com and downloads my.mov.
5. LP1 spawns a program to analyze my.mov and collects the results.
6. LP1 contacts the LDG and reports the statistics on my.mov and movie.html.

Each LP and LDG process runs on a separate UNIX workstation. In the experiment, eight machines ran the LP processes and one machine ran the LDG. It took about six weeks to download and analyze all 44,000 potential video

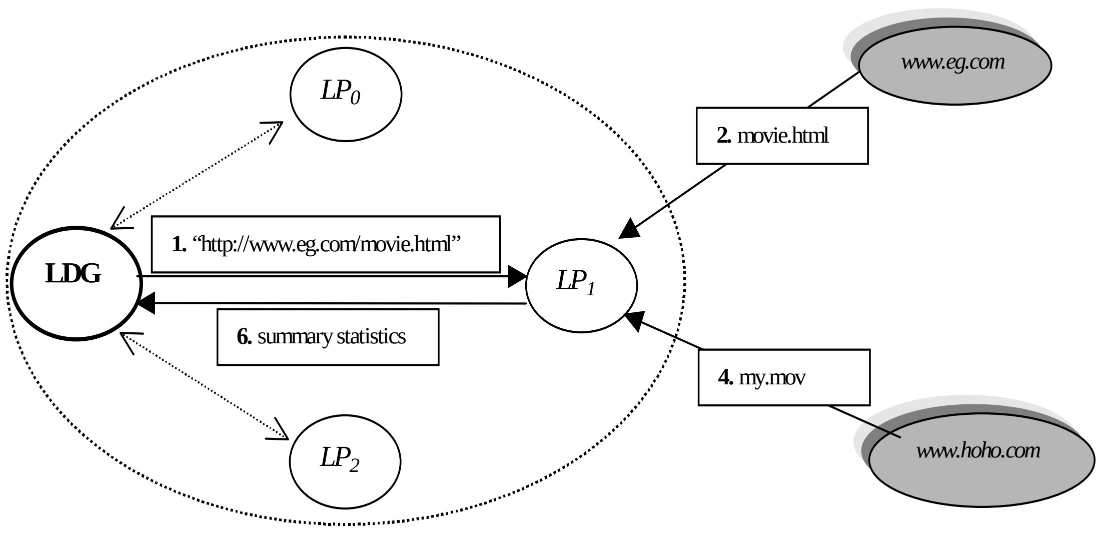


Figure 2.1: Video Link Processing Architecture (steps 3 and 5 occur within LP_1 and hence, are not shown)

documents. Of these, about 22,600 were valid links. The rest either did not exist or did not contain links to video data. Not surprisingly, link processing was faster at night due to lower Internet traffic. I wrote the core portions of the LP and LDG in Tcl, employing Tcl-DP [74] for communication between the LPs and LDG. I used *mpegstat* [66] to analyze the MPEG files, and an instrumented version of *xanim* [10] to analyze the QuickTime and AVI files.

About 10% of all the QuickTime, 8% of the AVI and 5% of the MPEG titles were not analyzed for one or more of the following reasons:

- There were QuickTime files that had not been "flattened"[1]
- xanim did not have the right QuickTime or AVI codec for the file
- The file was audio only
- The file had been truncated. A file was truncated if its downloaded size was not within 95% of the value claimed by the accompanying HTTP header.

2.2.3 Step 3: The Sifting Phase

In all, about 100 GB worth of HTML and video files were downloaded and processed, accumulating 25MB of raw statistics. From this initial list of about 57,000 titles, about 9,500 suspect videos were excluded using the following guidelines:

- *4 <= fps <= 40*.

  Files with frame rates less the four frames per second (fps) or greater than forty fps were excluded. I found many files to have a frame rate of 0.1 fps (some were zero), and others with a frame rate as high as 1000 fps. To

[1] “Flattening” is a process that combines the resource and data forks of a QuickTime file thus making it portable.

avoid skewing the derived characteristics, such as movie duration (frame count divided by fps), I eliminated these titles. This criterion eliminated about 5000 entries.

- *duration >= 0.5 seconds*

  Clips less than one-half second in duration were eliminated since they were too brief to qualify as "video." About 1000 links were eliminated this way.

- *0.6 <= AR <= 1.6667*

  The aspect ratio (AR = width/height) was constrained to be in this range to conform to acceptable norms for video. Spot-checking revealed that videos with aspect ratios outside this range were largely collections of images, rather than true motion video. Nearly 1000 links deviated from this guideline.

- *Bitrate < 10 Mbits/sec*

  I define the bitrate of a movie clip as:

  Bitrate = movie size (bits)/movie duration (seconds).

  I constrained the bitrate to be less than or equal to 10Mbits/sec. Files exceeding this limit were frequently images or simulations with very large sizes and very small durations, resulting in abnormally high bitrates. Almost 1000 files exceeded this threshold[2].

- Duplicate investigation of the same URL.

[2] The largest bitrate was 12.5 Gbits/sec.

To avoid downloading and analyzing the same URL more than once, the LP/LDG system checked a movie link against other already processed URLs before processing it. However, it did not account for common practices such as DNS aliasing where one machine can be referred to via multiple names. For example http://cnn.com points to the same location as http://www.cnn.com. Minimizing errors from this scenario involved comparing IP addresses, file names and sizes of all the processed URLs to detect duplications. 1500 titles were eliminated this way.

After completing this process, 47,500 titles remained. This working data set consisted of 53% QuickTime files, 30% MPEG files, and 17% AVI files.

2.3 Results and Analysis

I analyzed the collected data using Microsoft Excel and Tcl scripts. The raw data calculation fell into six types of categories, which are detailed in the following subsections:

- **Directly measurable quantities**, such as date of creation (section 2.3.1), frame rate (2.3.2), and movie size (2.3.3),
- **Derived quantities**, such as how average movie size is changing (section 2.4), movie duration (2.3.5), aspect ratio (2.3.6),
- **Codec properties**, which shows how the AVI (section 2.3.7), QuickTime (2.3.8), and MPEG (2.3.9) codecs are used to encode video for the Web,
- **Network properties**, which calculated the bandwidth required for streaming (2.3.10) and the pre-fetch buffer required to stream a movie without interruption (2.3.14),

- **Replication**, where I measured how many movies are replicated on the Web (2.3.11),
- **Age**, which analyzed the lifetime of the movies (2.3.12),
- **Geographical Origin**, where I investigated the distribution of all the movies by region (2.3.13).

2.3.1 Movie Date Distribution

A video file has two associated dates: its on-line date and document date. The on-line date of a video is the date it was placed on-line. The document date is the last modified time of the associated video document as reported by the Web server. For example, if an HTML page contains links to 10 MPEG files, the document date is the time the HTML page was modified, whereas the on-line dates are the dates when the MPEG files were modified.

Figure 2.2 plots the number of movies placed on the Web in a given time, using on-line dates. Figure 2.3 decomposes figure 2.2 into the three movie types. QuickTime is clearly dominant format today, although MPEG led until mid 1995. AVI, initially the least used of the three formats, is currently comparable to MPEG in popularity.

Figure 2.2 shows the growth to increase until May 1996, after which the growth levels-off and declines. This behavior raises two questions:

1. Why are there movies dated April and May 1997 when the cutoff date of our initial survey was March 1997?
2. Why is there a decline of movies coming on-line from December 1996 - May 1997?

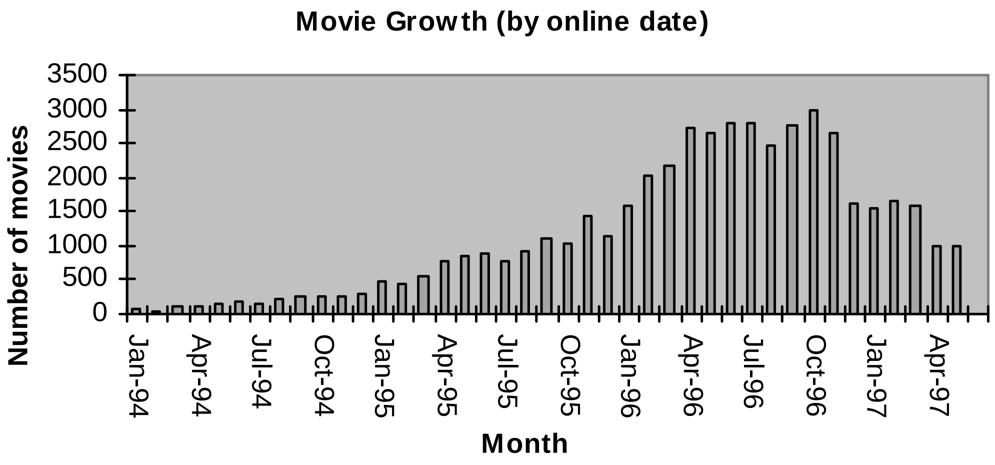


Figure 2.2: Movies Added To The Web

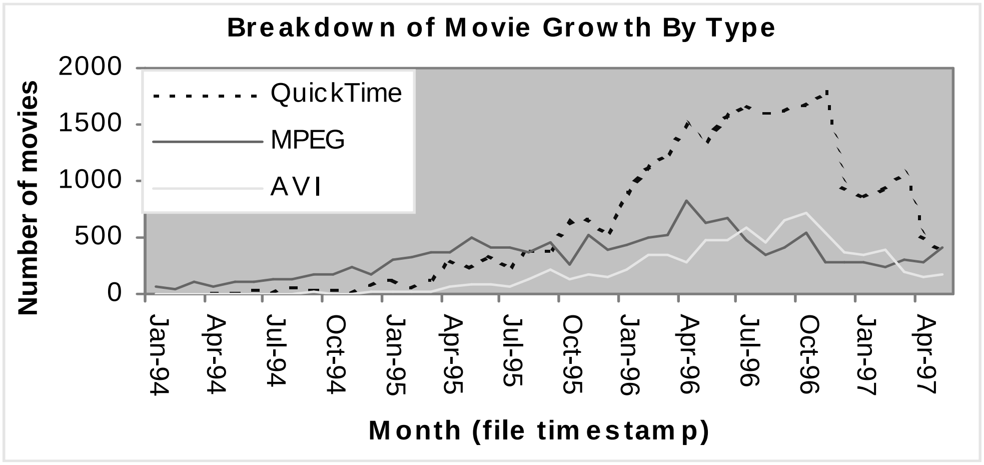


Figure 2.3: Number of movies added to the Web, by format

The answer to the first question is simply that the video documents were modified in the time period *after* the retrieval of the document URLs from Alta Vista and *before* they could be analyzed during the gathering phase. For example, many of the video documents are indices with dozens of movies. Suppose Alta Vista indexed such a document in March 1997, and that the author added a new movie in early May. That movie's on-line date would be correctly reported as May 1997.

There are a number of possible answers to the second question. These include:

- *Lag time in web indexing*: recall that the list of video documents was obtained from Alta-Vista. If we assume that Alta-Vista takes, say, six months to index the Web, then documents older than 6 months are certain to have been indexed. Documents that are more recent have a decreasing probability of being indexed. However, if Alta-Vista has not indexed the (HTML) video document, my search strategy would not find this video. For instance, if a video is placed on the Web in February 1997, but the associated video document is not indexed by Alta-Vista when we collect the list of potential documents, we will not find it. Thus, I am under-reporting the number of video files on the Web for dates close to March 1997.
- *The number of videos on the web is actually declining*: due to the recent introduction of streaming video technologies it may be possible that AVI, QuickTime and MPEG are no longer the encoding tools of choice for presenting movies on the web.

2.3.2 Frame Rate

AVI and QuickTime movies use low frame rates. Figure 2.4 displays the frame rate spectrum for all the movies in the data set. At the low end of the spectrum, there are peaks at 8, 10, 12 and 15. At the high end, most fps values cluster around 30 with smaller peaks at 24 and 25. The lower valued peaks in figure 2.4 are caused by QuickTime and AVI files, while the high peaks are largely due to MPEG files.

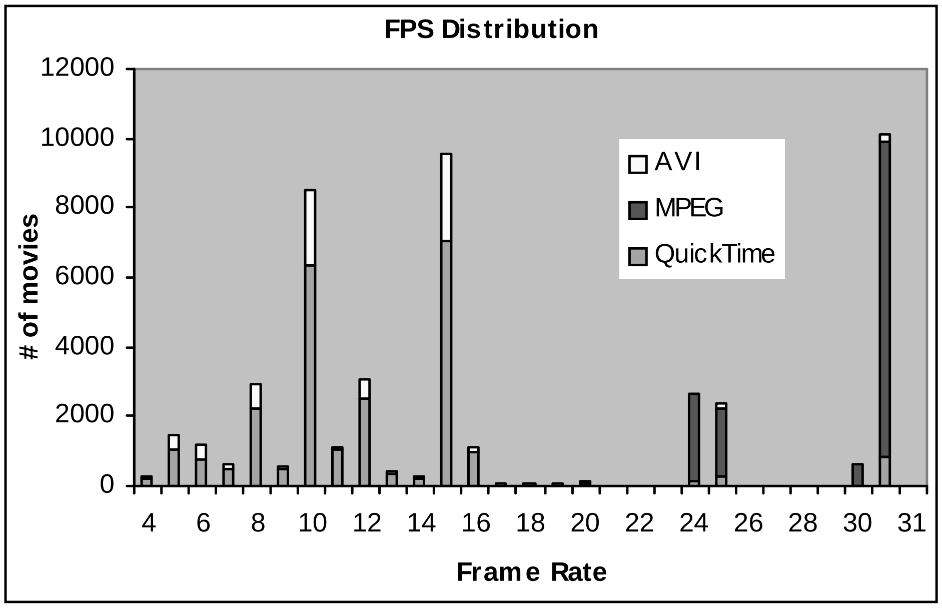


Figure 2.4: Frame Rates of video files

2.3.3 Size

Movie files are relatively small. Figure 2.5 plots the size distribution of all movies in our data set. It shows that 70% of the movies are 2 Mbytes or less. The median movie size is about 1.1 MB. Figure 2.6 breaks this distribution down by format. It shows that AVI and QuickTime files have similar size distributions, whereas MPEG files are smaller overall. As we shall see later, this characteristic of MPEG files can be attributed to their better compression and relatively smaller playback times.

2.3.4 Monthly Size

The median size of the typical movie is increasing, but the median size of a movie file of a given format is staying the same. Figure 2.7 plots the mean and median size of movie files versus time across all movies (top) and by movie type (bottom). Each data point represents one three-month period, except the first

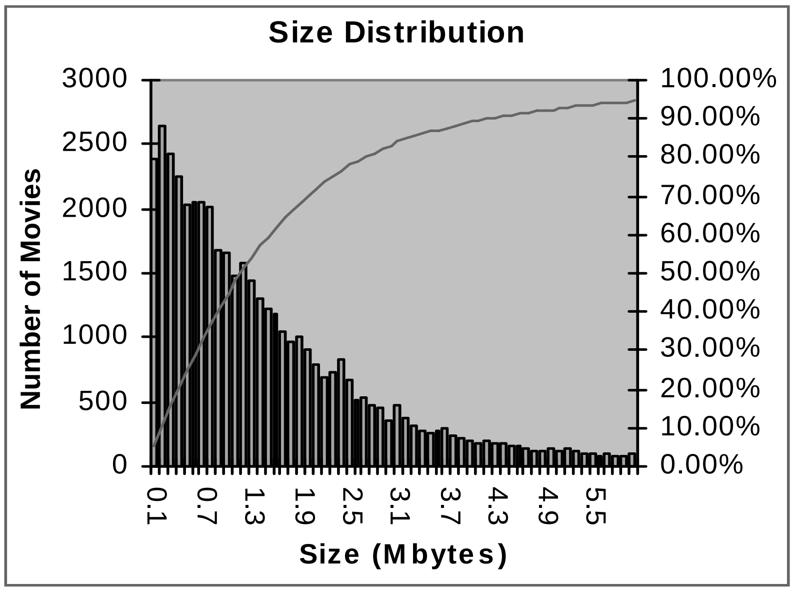


Figure 2.5: Size Distribution of Video Files

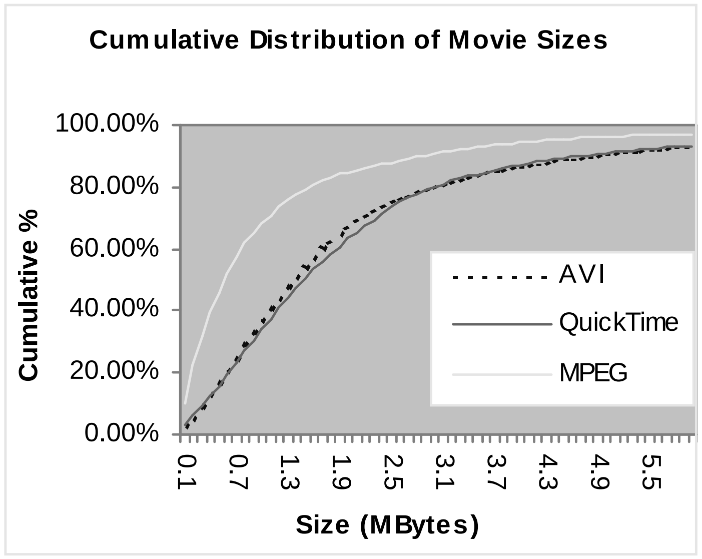
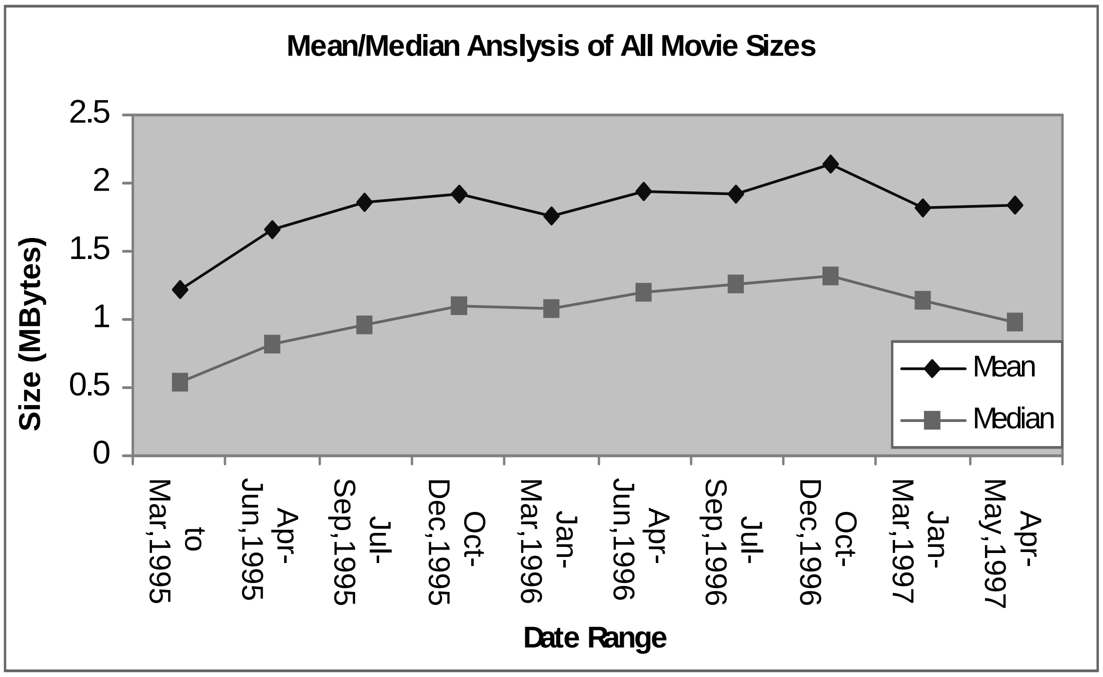


Figure 2.6: Size Distribution by type

point, which represents a fifteen-month period. The expansion of the first time segment was required to provide sufficient data points. As the top figure shows, the median size is clearly increasing. The bottom figure just as clearly shows that the median size of MPEG and QuickTime movies is remaining constant.



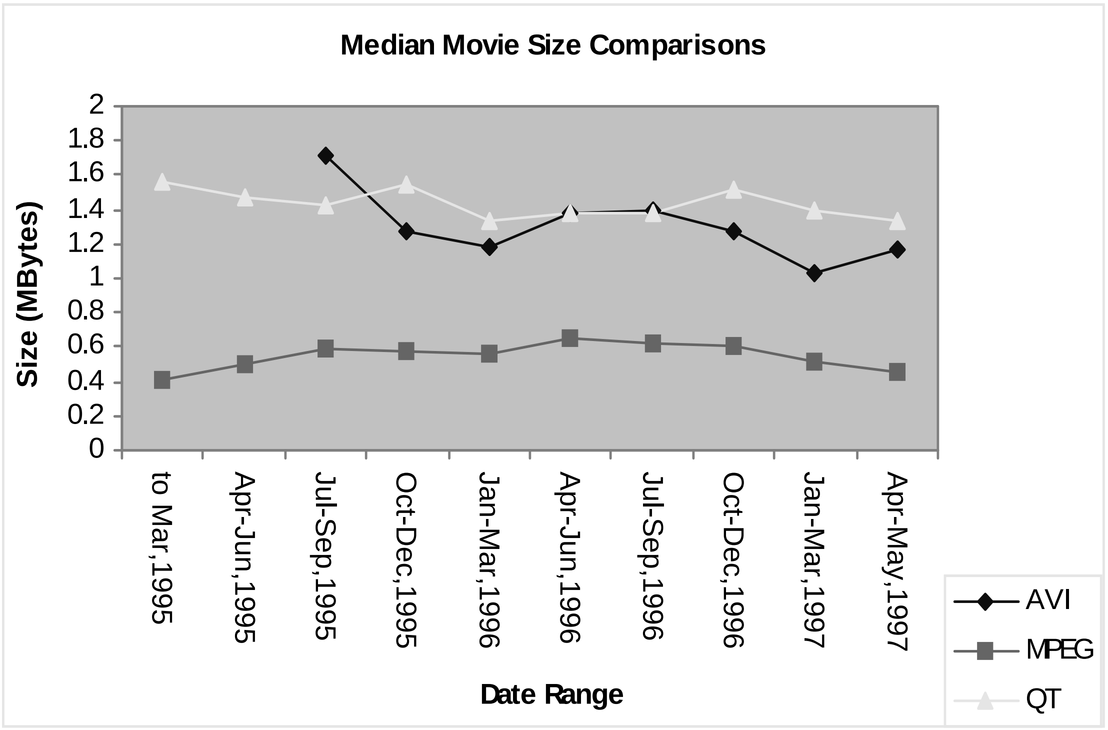


Figure 2.7: How Movie Size Is Changing, overall (top), by type (bottom)

The reason for the rise in the top figure is the increase in popularity of Quick-Time (figure 2.3). Since QuickTime movies are generally two to three times larger (in bytes) than MPEG movies, as shown in the lower figure, a high percentage of QuickTime movies drives the average up. The drop in the popularity of QuickTime in 1997 accounts for the decline in the last two quarters of the median and mean movie size in the top graph.

2.3.5 Duration

Movies on the Web are short. I calculated the duration of a movie by dividing the number of frames in the movie by the frame rate. Figure 2.8 shows the number of movies of a given duration, and figure 2.9 breaks down figure 2.8 by format. 90% of movies are 45 seconds or less in duration, and half of the movies were fifteen seconds and under. The right-hand figure highlights another interesting result: MPEG files are generally shorter than their AVI/QuickTime counterparts. The latter two formats have almost identical duration distributions.

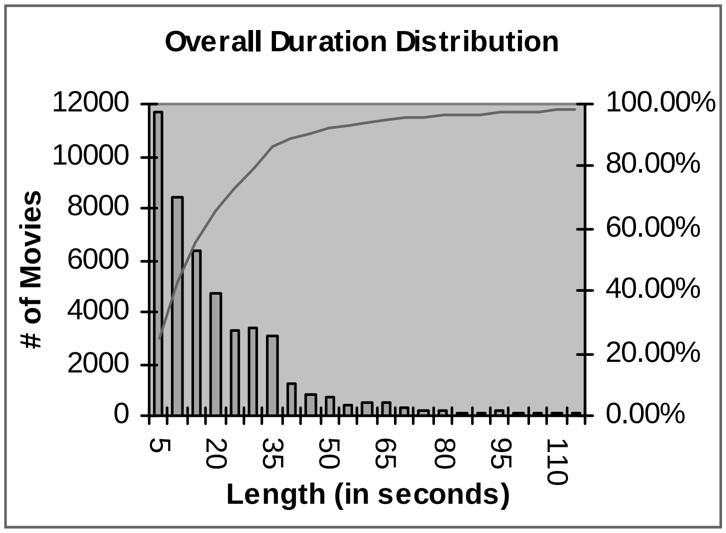


Figure 2.8: Duration of all movies

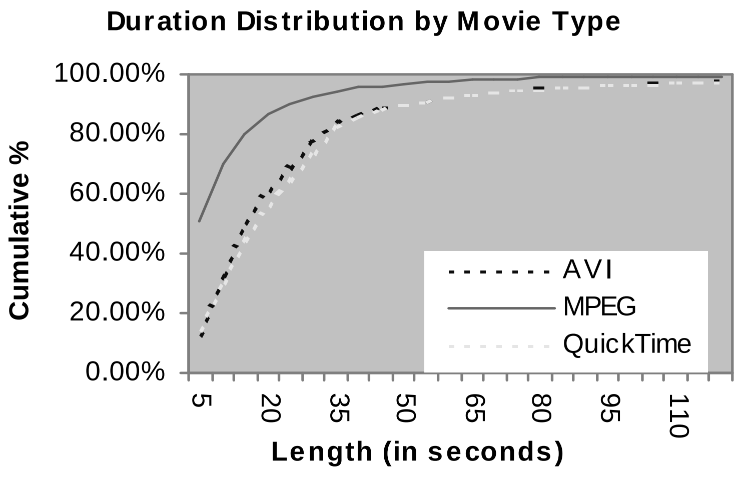


Figure 2.9: Duration of all movies by type

2.3.6 Aspect Ratio

74% of all files have an aspect ratio (width over height) of 1.33, which corresponds to a movie size 160x120 and 320x240. 15% of the remaining files have aspect ratios ranging in between 1.2 and 1.5.

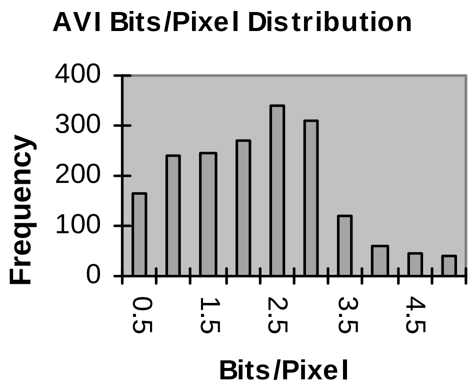
2.3.7 AVI

About 25% of all the AVI files had no audio. 90% of the audio/video files used PCM as their audio codec. Radius Cinepak was the most popular video codec (43%), followed by Microsoft Video 1 (26%) and Intel Indeo R3.2 (25%).

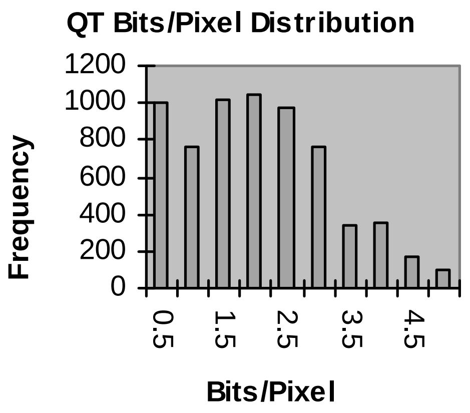
I used the bits/pixel metric to analyze video compression performance:

bits/pixel = video size (bits)/ (width height* number of frames)*

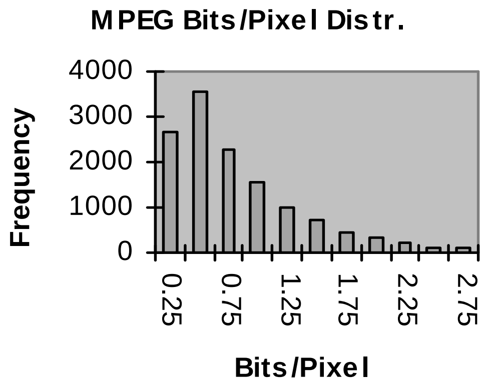
I computed the metric on video-only files and figure 2.10 (top) displays the resulting distribution. The mean bits/pixel was 2.51 and the median was



AVI mean bits/pixel = 2.51,
median = 2.14



QT mean bits/pixel = 2.16,
median = 1.82



MPEG mean bits/pixel = 0.73,
median = 0.53

Figure 2.10: Typical compression performance of formats

2.14. Both Radius Cinepak and Indeo had similar mean bits/pixel performances at around 2.0 bits/pixel and Microsoft Video was slightly worse at 2.4 bits/pixel.

2.3.8 QuickTime

About a third of the QuickTime files were video only. PCM was again the dominant audio codec for audio/video streams (84% of all the a/v QuickTime files). Figure 2.10 (center) details the overall bits/pixel distribution. Although it is similar to AVI, QuickTime compresses slightly better with a mean bits/pixel value of 2.16 and median of 1.82. The most popular video codecs were Radius Cinepak (60%) with a median bits/pixel of 1.9 and Apple Video-RPZA (22%) with 2.6 bits/pixel. I found the best video compression to come from the JPEG codec (6% popularity) which had a median bits/pixel of 1.6.

2.3.9 MPEG

Figure 2.10 (bottom) illustrates that MPEG's compression is superior to that of QuickTime or AVI, since the bits/pixel distribution is more concentrated in the low bits/pixel range. I found the MPEG files to have a mean bits/pixel value of 0.73 and a median of 0.53. Only 7% of MPEG files had audio, in contrast to QuickTime or AVI. The lack of MPEG files with audio is probably due to the fact that early MPEG encoders were video only. Recently created MPEG files tend to have audio.

Table 2.1 provides the statistics on individual MPEG frame types: P frames compress about twice as much as I frames, and B frames compress by a factor of 5 better than I frames.

Table 2.1: Frame Type Analysis

Frame Type	**Mean bits/ pixel**	**Median bits/pixel**
I	1.25	1.10
P	0.76	0.54
B	0.31	0.19

Investigating the frame patterns in MPEG streams showed that about 80% of all MPEGs had some type of repeating frame pattern. Table 2.2 shows the various patterns (in playback order) and the corresponding mean bits/pixel. The pattern of I frames only recurred most often followed in popularity by the sequence IBBPBB. Note that the bits/pixel value drops when more B frames, relative to P and I, are in the pattern,. The presence of common frame patterns indicates that MPEG users are content to use the default values in their encoders.

Table 2.2: Frame Pattern Distribution

frame pattern	**% distribution**	**mean bits/pixel**
I	27.10%	1.17
IBBPBB	15.70%	0.7
IBBPBBPBBPBBPBB	10.40%	0.31
IBBPBBPBBPBB	8.10%	0.5
IBBBPBBBPBBB	4.40%	0.66
IPBBIBB	4.20%	0.39
IIP	3.50%	0.7
IBBBPBBBPBBBPBBB	2.90%	0.58
IPBB	2.00%	0.62
IPBBBPBBBB	1.90%	0.28
IPBBPBBPBBPB	1.20%	0.51
IPPP	1.20%	0.79

2.3.10 Bandwidth Requirement

The Internet today is incapable of streaming most MPEG, QuickTime, or AVI video stored on the Web. Figure 2.11 shows the average bitrate distribution, calculated as the movie size (in bytes) divided by its duration (in seconds). As we can see, at basic ISDN rates (around 128 Kbps) only 3%-4% of the titles can be streamed. At 700 Kbits/sec, 52% of all movies can be streamed, and at 1.5

Mbits/sec 84% can be streamed. Clearly, a disparity exists between bitrates achieved by established compression technologies and current modem speeds. It is also interesting that despite the pressure to make the viewing of MPEG, QuickTime, and AVI video tolerable over slow connections, authors seldom drop below 500 Kbps when creating their content. This suggests that this bitrate represents a lower limit in quality for these codecs. Below this bitrate, the quality is simply unacceptable.

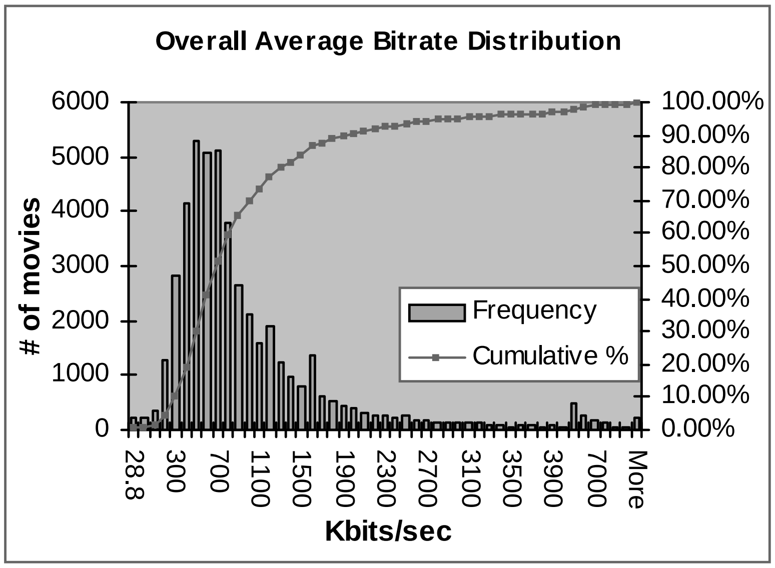


Figure 2.11: Bitrate requirements for streaming video

To investigate bitrate distribution further, I defined the property of *transferability*: a file is transferable at a certain bandwidth if its average bitrate is at or below that bandwidth. I first classified the movie collection by type and, within each type, subdivided further depending on whether the movie was video-only or had both audio and video. I then calculated the transferability, at various bandwidths, of the files in each category. Table 2.3 itemizes the results.

Table 2.3: Comparison of BW

Bitrate	QT	QT w.o. audio	AVI	AVI w.o. audio	MPEG	MPEG w.o. audio
28,800	0%	2.70%	0%	0.20%	0%	0.06%
56,000	0%	5.55%	0%	0.99%	0%	0.19%
200,000	0.69%	12.26%	1.44%	9.34%	1%	6.35%
600,000	43.28%	42.39%	41.58%	38.22%	36%	37.58%
1,500,000	91%	79.02%	84.73%	78.28%	87%	75.51%
5,000,000	99.77%	96.67%	99.65%	97.42%	97.30%	95.43%
10,000,000	99.97%	99.33%	99.95%	99.25%	99.20%	98.91%

We observe two main points:

1. QuickTime was generally more transferable than either AVI or MPEG, and MPEG was the least transferable. I hypothesize that this is due to its high frame rates, which raise its average bitrate.
2. Within each format, audio/video streams are more transferable than their video only counterparts at the higher bandwidths. At first, I found this effect counter-intuitive - I expected the presence of audio to raise the average bitrate, not lower it.

To investigate the cause of observation 2, I plotted uncompressed bitrate of each video stream:

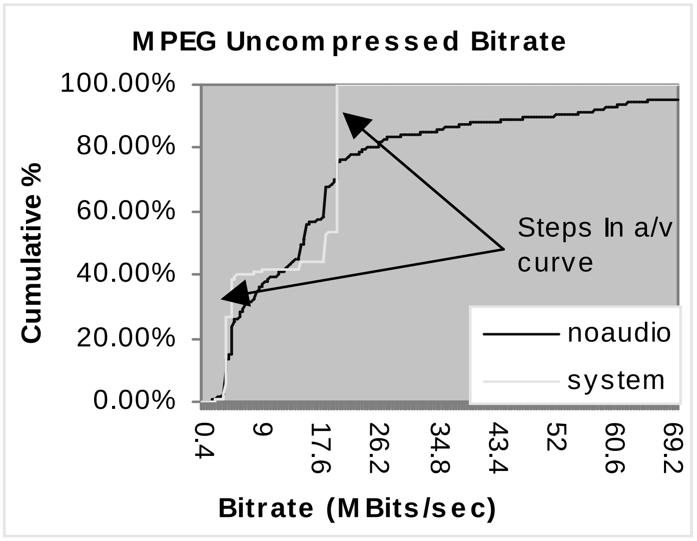
$$U = 8 * \textit{video width} * \textit{height} * \textit{fps}$$

Figure 2.12 (top) plots the cumulative distribution of U for MPEG video-only and audio/video files. It shows, for example, that 40% of MPEG audio/video files have $U \leq 6.5$ Mbits/sec, and almost all have $U \leq 20$ Mbits/sec. Figure 2.12 (bottom) plots the same metric for QuickTime. The steps in the audio/video curves are caused by files created with common picture dimensions and frame rates. For example, the MPEG systems file curve in figure 2.12 (top) has a large step at the 5 Mbits/sec region and another around the 20 Mbits/sec region. The magnitude of the steps indicates that the majority of the files are located around these regions. The first step is due to 160x120 files (30 or 25 fps), and the second step is caused by files 352x240 in dimension (30 fps) and 352x288 in dimension (25 fps). The plot of QuickTime files shows a similar pattern. Here the steps in U for audio/video files occur around the 2.4 bits/sec (160*120, 15fps), 2 Mbits/sec (160*120, 12 fps) and 1.6 Mbits/sec (160*120, 10 fps) regions. The AVI uncompressed bitrate distribution is very similar to that of QuickTime, and therefore not shown.

In contrast, video-only files have no such strong characteristics. They have a large variety of shapes and sizes, as reflected in the distributions of U shown in figure 2.12. Spot checking of the video files indicates that this is because video-only movie files are often used to present the output of simulations and computer animations, which do not have the size restrictions of NTSC or PAL video, the typical source for audio/video data.

2.3.11 Movie Replication

My criteria for considering a movie to be a copy of another was as follows: if a movie on a different WWW server had the same size, type, width, height, and fps I considered the movie to be replicated. Although this does not directly compare the two movies, it is similar to comparing a checksum. Random testing showed this criteria to bereliable. I found 2177 unique movies that had been replicated. The majority of movies were replicated once with the maximum being 53. Popular replicated movies included "standard" reference files such as "bike.mpg", "moglie.mpg" and "RedsNightmare.mpg".



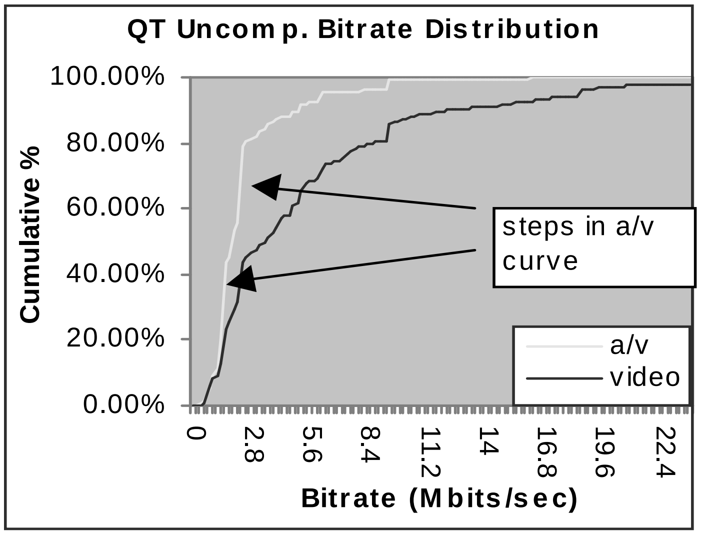


Figure 2.12: Uncompressed bitrate for MPEG (top) and Quick-Time (bottom)

During the gathering phase of my survey, I was careful to check URLs for duplicates to avoid unnecessary processing and downloading. However, this particular analysis revealed a potential gray area: how to differentiate between movies present on the same WWW server with identical replication characteristics but with different path names? For future analyses, I plan to ignore such duplicate instances.

2.3.12 Geographical Origin

An obvious approach to classifying movies according to origin would be to look at the lifetimes extension of the DNS name of the server hosting the URL. Non-USA machines have extensions in their names that indicate their country of origin and it would be a simple matter to parse and categorize the URLs accordingly. However, DNS aliasing can cause machine names that appear to originate from the USA may in fact be situated elsewhere. For example, the machine name *www.dotmusic.com* is in fact an alias for *www.dotmusic.co.uk*. To reduce the problem, I used the UNIX *whois* command (which accesses the InternNIC services [96]) to verify that the postal address registered for every US domain name from the working data set did in fact originate from the USA. If not, I altered the data set to reflect the real country of registration. This solution is not comprehensive since large organizations such as ISPs, ostensibly registered to USA addresses, have networks that span continents. Nevertheless, the findings as shown in figure 2.13 do illustrate the current trends in physical location of movies. In figure 2.13, movies are classified according to broad physical regions and it is not surprising to see that the majority of the movies in our data set are located in North America with Europe being the next largest. The “India and Asia” region follows next in popularity but in reality, the majority of movies from this part of the world are situated in Japan. Not surprisingly less developed countries in Africa, Central America and Pacifica (islands in the Pacific Ocean) are the regions with the least number of movies.

Figure 2.14 illustrates the type composition of movies in each region. Most of the time, the pattern is similar to that observed in section 2.1: QuickTime is dominant with MPEG and AVI following. The only exception is Europe where

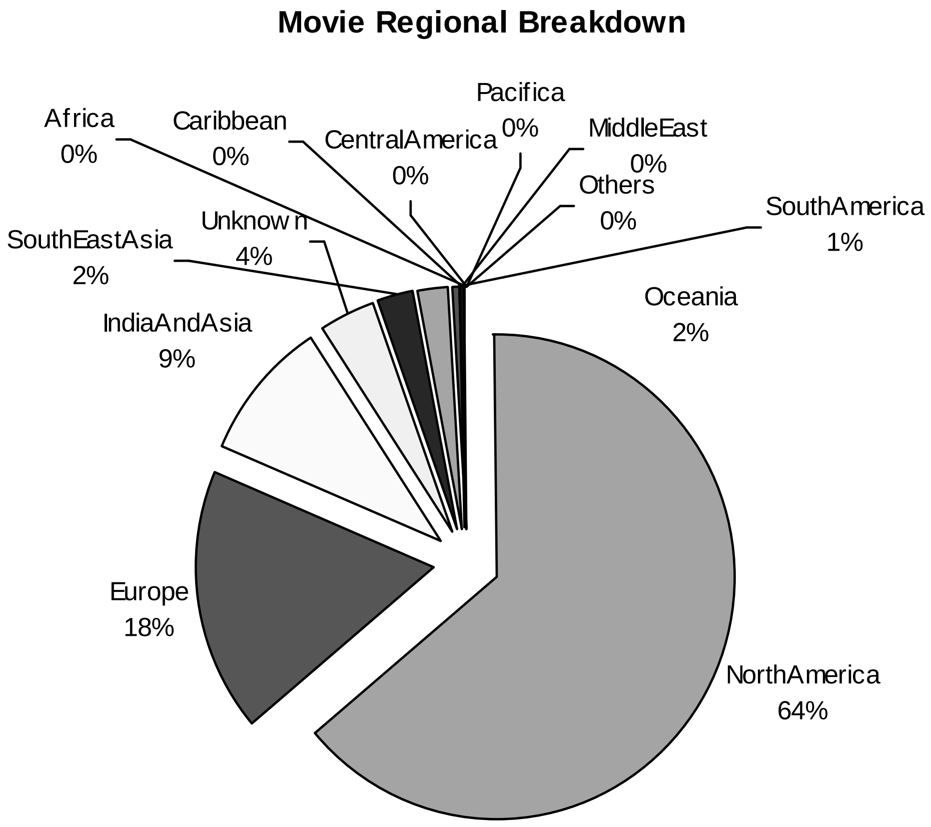


Figure 2.13: Breakdown of movies by region

MPEG leads over the other two formats. This is probably because European researchers have been more active than their American counterparts in standardizing and adopting MPEG.

2.3.13 Movie Age

Do movies last with time? Four months after the original survey, I rechecked the links from the working data set to verify whether they were still at their original location or not. The process involved performing a “HEAD” query at the WWW server given by the original URL and, if the video was still there, comparing the recorded size of the movie with that returned by the query. I ran the verification process twice. The intention of the second iteration was to track down videos that had changed locations (the first iteration indicated that 1% of

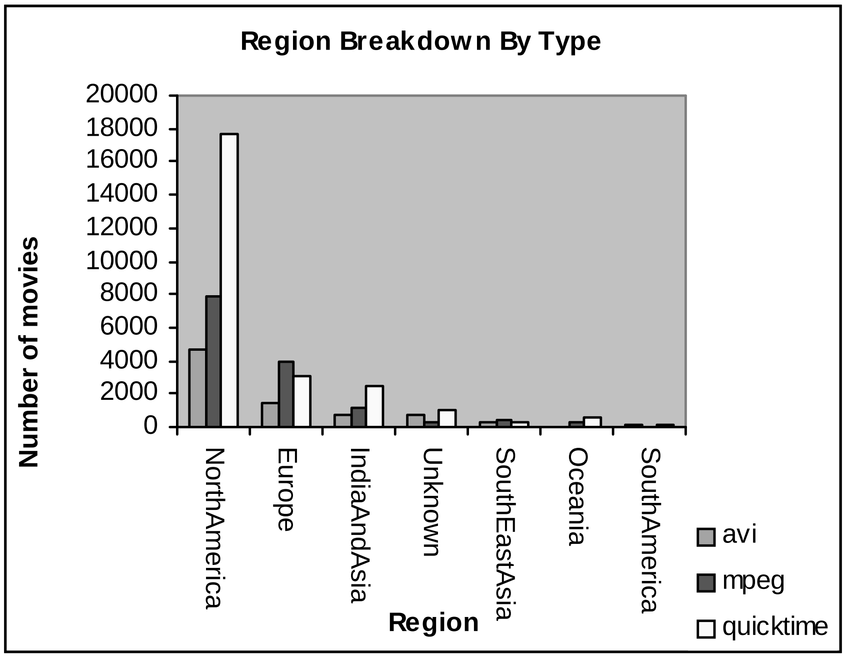


Figure 2.14: Breakdown of movies by region and type

the documents had moved) and to retry sites that could not be accessed in the first iteration. Table 2.4 summarizes my findings.

As can be seen, the majority of files (80%) were accessible confirming

Table 2.4: Age Analysis Results

Error Reasons (if any)	**% of documents**
HTML code 200 (OK)	80.4%
File size changed	0.7%
HTML code 404 (Not Found)	12.9%
HTML code 406 (None Acceptable)	2.3%
Network Errors	3.2%
Other Errors	0.5%

my hypothesis that video data on the web tends to be write-once-read-many in nature. Random checking of the “file size error” reports indicated movies of this type tended to be animated weather maps or satellite pictures that required a constant name yet were updated on a regular basis. “Network errors” were primarily due to a site being down, permanently or otherwise.

2.4 Related Work

Woodruff [61] and Bray [10] have inspected WWW content. They have looked at as much of the Web as possible in order to characterize document sizes, HTML tag usage, file types used as URLs, and so on. Their results indicate that videos do not account for a very significant portion (less than 1%) of WWW content. Once again, their analysis is based on data collected during 1995. However, according to Woodruff, MPEGs comprise the highest proportion of video files (0.3%) during this time, followed by QuickTime (0.2%) and AVI (0.1%) respectively. This particular order agrees with our findings of video on the web for the 1995 time period.

The study by Smith and Chang [56] is, to my knowledge, the only previous work that has analyzed video on the web. They have implemented a system for traversing the web that locates and indexes images and videos. Their focus is on merging text-based processing and content-based visual analysis to produce an easily searchable database. Videos form a small portion of the data they have gathered (about 1%). My approach concentrates on the direct analysis of videos and how they integrate into the World Wide Web. Our data is unique since it is the first large-scale study of its type.

2.5 Conclusion and Future Work

It is clear existing compression technologies do not provide low enough bitrates for streaming transmission over standard modems. One solution is to raise the network bandwidth, and my study indicates that 700 Kbits/sec to 1.5 Mbits/sec is an appropriate value. However, in the absence of sufficient network bandwidth, it is interesting to observe how user behavior and video technologies have evolved to address these problems.

One approach is to improve video compression performance [76, 77, 79, 95] thus reducing the required bandwidth.

Users of MPEG, AVI or QuickTime are attempting another approach to the bandwidth problem by creating files relatively small in size and duration (when compared with their VHS counterparts). However, authors have not throttled the bitrate of the videos at the expense of picture quality. This implies they are not willing to sacrifice video quality for bandwidth - there is a perceptual threshold below which authors are unwilling to descend. The corollary of this is that every video technology has some sort of critical bandwidth associated with it - users cannot tolerate the picture quality for videos encoded below this bandwidth.

For the purposes of this thesis, the result that video files tend to have the write-once-read-many property is crucial, allowing the investigation of an alternate cache based approach to transmitting videos over the Internet. I also use the average file size and other properties discovered in this chapter for simulation purposes. This work is presented in chapter 6.

In the future, I plan further investigation of video movie distribution. The need for an additional study is motivated by the following reasons:

- It would clarify the reason behind the drop in the number of movies coming online, as described in section 3.3.1. It would be useful to see what kind of patterns emerge in the second study. In particular, it would be interesting to see if the number of streaming videos coming online has increased or not.

- It would investigate the properties of streaming videos. However, extracting meaningful information out of a streaming video file encoded in a proprietary format may be impossible.
- It would confirm the general video characteristics uncovered in the first study.
- It would enlarge the scope of the study. The first study relied on querying Alta Vista for video documents. Getting such data directly from one of the sites that operate web crawlers would present a clearer and more comprehensive picture of videos on the web.
- The results of the geographical analysis is still suspect. Better methods are necessary for determining the origin of an URL.
- It would be interesting to see how many clips are replicated in multiple formats and to develop techniques for detecting such replication.
- Investigation of MPEG-2 files.
- It would be useful for the general multimedia research community if this sort of study was carried out on a regular basis. Doing a second study would provide much insight as to the additional tools and facilities needed to automate the process. The tools developed would be of value themselves.
- Local storage of video data for future analysis: if the total size of the number of movies on the Internet is indeed on the order of hundreds of Gigabytes as the first survey suggests, it may be feasible to download and store all of them, if not to hard disk, then at least to tape. Availability of the movies will allow further verification of any results derived from the data made and allow easy deployment of new analysis techniques.

Chapter 3

Video Trace Analysis

3.1 Introduction

In the previous chapter, I analyzed characteristics of video data stored on the web. However, such an analysis presents a partial picture of video usage: we know about the properties of the video files but we do not know how these files are accessed. Knowledge of access patterns can be useful to MiddleMan design. For instance, if access patterns revealed that smaller files were more likely to be accessed than larger files, the cache replacement policies of MiddleMan could be optimized to retain small files.

In this chapter, I study user access patterns and file characteristics of an ongoing VOW experiment in Luleå University of Technology, Sweden. This VOW experiment is unique because video material is distributed over a high bandwidth network. Hence, users can make access decisions without the network being a major factor. Similarly, the data being transmitted is also created and placed online without regard to network conditions. Given that my study of videos on the web in the previous chapter indicates that currently content creators, faced with large Internet latencies, deliberately throttle video sizes and durations in an attempt to reduce download wait times, the Luleå analysis provides insights into the potential behavior of both users and content creators in the Internet of the future where bandwidth is more plentiful. Sample findings included:

1. *Inter-arrival times*: median interarrival time of about 400 seconds indicate that requests for videos are nowhere near as frequent as those for HTML documents.
2. *Video browsing patterns:* users like to view the initial part of videos in order to determine if they are interested or not. If they like it, they continue watching. Otherwise, they stop.
3. *Temporal Locality:* accesses to videos also exhibit strong temporal locality. If a video has been accessed recently, chances are that it'll be accessed again soon.

The remainder of this discussion is organized as follows. Section 3.2 provides background information on the VOW experiment and section 3.3 presents an analysis of video file characteristics. In section 3.4, I detail my criteria for eliminating erroneous data from the file access trace and, in section 3.5, I present specific results from my analysis. I conclude with some observations in section 3.6.

3.2 Background

Since 1995, the Centre for Distance-spanning Technology at Luleå University has been researching distance education and collaboration on the Internet [97]. Specifically, it has developed a hardware/software infrastructure for giving WWW-based courses and creating a virtual student community. The hardware aspects include the deployment of a high speed network (2-34Mbps backbone links) to attach the local communities to the actual University campus. The campus is also connected to the national academic backbone by a high speed 34 Mbps link [45] with student apartments being wired together with the rest of campus via 10 or 100 Mbps ethernet. On the software side, the mStar environment, developed at CDT, provides a collection of web-based authoring, presentation and recording tools that use the Mbone for content delivery [20]. In particular, the mMOD (the multicast Media On Demand) system, a component of mStar, allows for recording and playback of classroom lectures, seminars and meetings [44].

The mMOD system consists of two separate programs, the VCR and the Web Controller. The VCR allows for recording and playing back of broadcasts. H.261 [59] is used for video compression in most cases. Recorded data is stored on the mMOD web server. The VCR also permits the fast forwarding or rewinding of a video stream.

The Web Controller provides an interface for the mMOD system - it allows users to request new video/audio playback sessions from the mMOD server. Additionally, it also permits users to join sessions already in progress. Users are able to view material via standard tools such as vic or vat [34]. A set of Java applets are also available for this purpose [98].

Figure 3.1 shows the playback architecture of the mMOD system on the Luleå University campus. User requests arrive at the mMOD server from three main subdomains within the campus, as well as from external sources. Since the mMOD server is the focus of both recording and playback, its log files form the basis for video access analysis, while its file system records provide the raw data for determining intrinsic file characteristics.

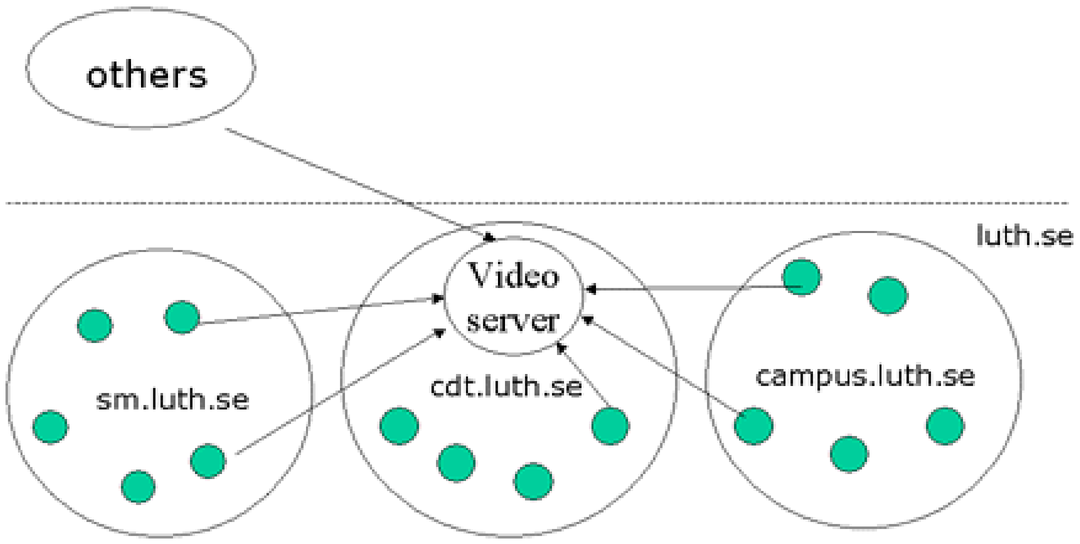


Figure 3.1: Video access structure on the mMOD system

3.3 Video File Characteristics

As of 10th March, 1998, the mMOD server (mmod.cdt.luth.se) hosted 139 audio/video titles that, according to the log traces, had been accessed at least once. Video content ranged from classroom lectures and seminars to traditional movies. Student enrollment in the undergraduate courses ranged from 100-140 with smaller numbers attending the graduate courses. All movies were CIF (320 x 240) in size. In the remainder of this section, I outline the basic characteristics of these file - their size, durations, and bitrate distributions.

3.3.1 Size

The file size analysis was based on detailed directory listings from the mMOD video server. Overall, the files totalled 15.7 Gbytes in size. Individually, each title is composed of separate audio and video files but in this analysis I aggregate them together. Figure 3.2 shows the individual size distribution of titles. 125 Mbytes is the most common file size and the mean value is about 121 Mbytes. In general, file sizes were several orders of magnitude larger than videos on the web.

3.3.2 Duration

Since no separate record exists about the duration of the titles, I determined the length of each by hand. This involved fast forwarding each title to its end via the mMOD VCR and noting the time elapsed. Some of the titles did not have accurate embedded timestamps and I ignored those for this analysis. Figure 3.3 displays the duration distributions of all the remaining titles. The distributions varied widely - from 10 minutes to over two hours. 90-100 minutes proved to be the most popular time range, most probably because this was the average length of a class lecture. The mean duration was approximately 75 minutes. Once again, the duration of these movies were much larger than those reported by the study of videos on the web.

3.3.3 Bitrate Distribution

To obtain the mean bitrate for each movie, I divided the size by its duration. Figure 3.4 plots the resulting bitrate distribution. The majority of the files exhibited bitrates between 150-250 kBits/sec, much lower than expected. This was because the video quality of each transmission was deliberately kept low [45] in order to save bandwidth for county viewers outside the campus with low bandwidth network access. Additionally, H.261, the video compression scheme used for the bulk of these streams, is mainly designed to produce low bitrates.

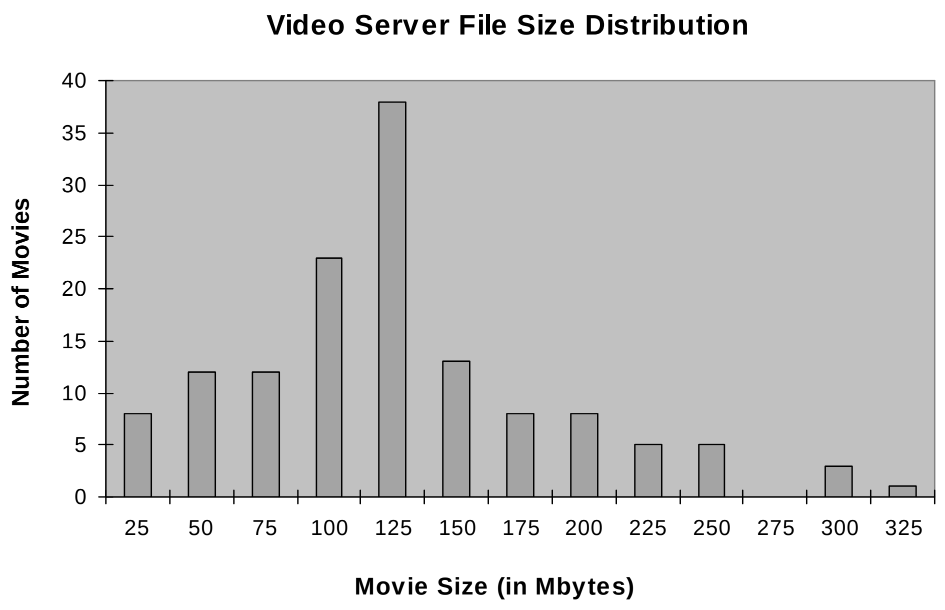


Figure 3.2: Video Server File Size distribution

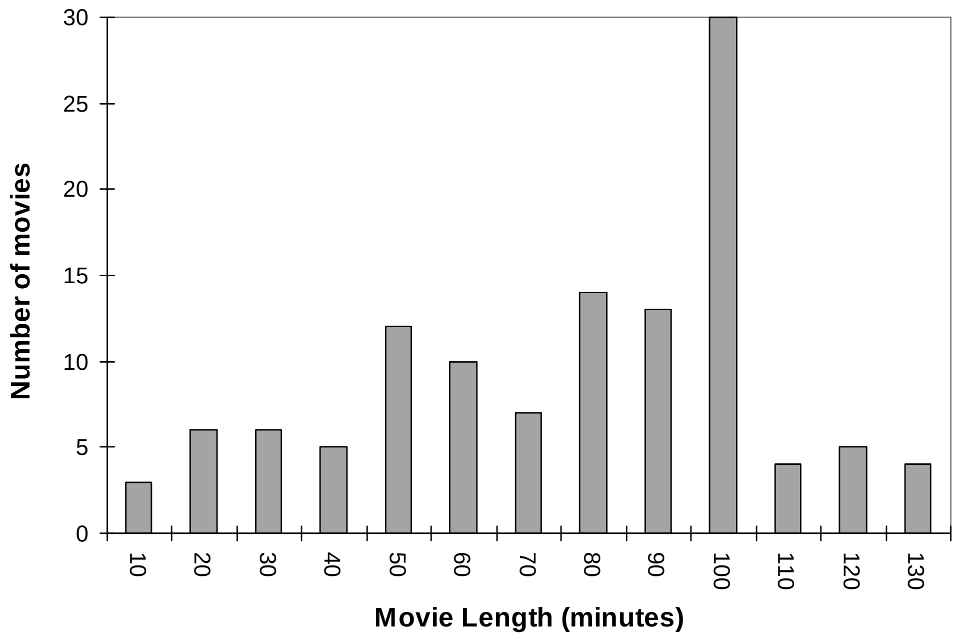


Figure 3.3: Movie Duration Distribution

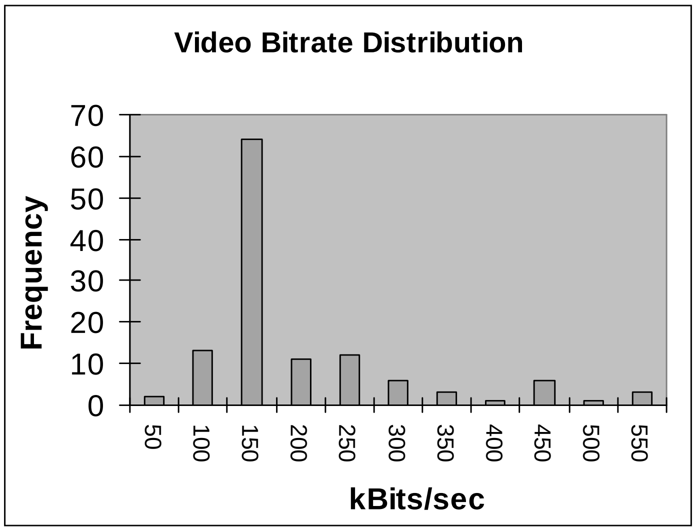


Figure 3.4: Video Bitrates

3.4 Trace Access Analysis

The trace that I analyzed was derived from the logs of the mMOD video web server. These logs recorded accesses to mMOD files from 29th of August, 1997 to 10th March, 1998 - little more than six months. My first step, the removal of excess and erroneous requests from this raw data, is reported in section 3.4.1. I divide my subsequent investigation of the refined data into two broad parts:

- **General:** How do video requests vary by day (3.4.2)? Do accesses to movie titles follow any specific mathematical distributions (3.4.3)? Do some machines request more often than others (3.4.4)?
- **Pattern Detection:** Here, I searched for invariants in the data. Are there any patterns in inter-access times of user requests (3.4.5)? Do users view titles all the way through or do they stop beforehand (3.4.6)? Do accesses vary

depending on the type of file (3.4.7)? Do requests exhibit any degree of temporal locality (3.4.8)?

3.4.1 Initial Log Filtering

The requests logged by the mMOD web server can be classified as follows:

1. Starting a video playback session: the user requests the setup of the appropriate video transmission environment (a multicast/unicast group) for a certain file. Playback starts from the beginning of the file by default - however the VCR applet can be used to fast-forward or rewind as necessary.

2. *Stopping a session:* the web server halts transmission and removes the multicast/unicast group dedicated for the transmission of this file.

3. Joining a session already in progress: the user joins the multicast group devoted for the transmission of this title.

4. Obtaining HTML documents.

5. Retrieving images.

Of these, I eliminated types 3-5 from the initial logs. Session joins (3) were too few to be statistically significant and event types 4-5 were irrelevant to my study. The distilled log consisted of the following entries:

```
<timestamp> <machine-name> <command> <title>
```

<*timestamp*> was the time, in seconds, when the request was made. <*machine-name*> indicated the originating machine. <*command*> was either `GET` or `STOP`, depending on whether the user wanted to commence or halt a

video playback session. Finally, <*title*> gave the name of the movie desired. A sample log is shown below:

```
872874233 salt.cdt.luth.se GET Movie1
872875354 spock.cdt.luth.se GET ArkivX_970206
872876661 aniara.cdt.luth.se GET Movie2
872876743 aniara.cdt.luth.se STOP Movie2
```

After this initial cleanup, I performed further filtering on the simplified trace, including:

1. Eliminate all requests from a particular machine which had been used for demo purposes and hence would have had unusual access patterns.
2. Remove dangling STOPs caused by the user hitting the STOP button too many times.
3. Some machines in the trace were only identified by their IP addresses. I replaced the IP addresses by their symbolic names.
4. Ignore consecutive `GET` requests from the same machine for the *same* movie if they are within 20 seconds of each other. For example, in the case of:

   ```
   02:01:01 aniara.cdt.luth.se GET Movie3
   02:01:15 aniara.cdt.luth.se GET Movie3
   ```

   the time difference between two requests is 14 seconds, hence the first request is ignored. The assumption is that there were problems in getting the first request to run and that is why the user started another request for the same movie. For time gaps more than 20 seconds, I assumed that the user genuinely wanted multiple streams of the same movie possibly because the same machine had multiple users or the user was editing this particular title.

If, on the other hand, the first request had a corresponding `STOP` like the following:

```
02:01:01 aniara.cdt.luth.se GET Movie3
02:01:10 aniara.cdt.luth.se STOP Movie3
02:01:15 aniara.cdt.luth.se GET Movie3
```

then both requests were acceptable since the user had deliberately stopped the first request.

Steps 1 and 2 eliminated about 300 playback requests leaving 5249 accesses overall. I carried out my subsequent analyses on this trace.

3.4.2 Video Access Grouped By Day

Figure 3.5 plots six months worth of server access grouped in 24 hour periods.

Figure 3.5 shows a cyclically fluctuating pattern of access which, with the exception of days 119-130, gradually increased with time. I found that accesses dropped off during weekends and rose again during the weekdays. Days 119-130 coincided with Christmas vacation when activity was minimal. Finally, the number of accesses increased significantly post Christmas. This was probably due to widespread deployment and usage of the mMOD system during the new semester, especially after the initial bugs had been ironed out.

3.4.3 Video Accesses To Movie Titles

Previous research on WWW traces [18] have shown that accesses to web documents tend to follow a Zipf distribution. Zipf's law [65], as applied to web access, states that given a collection of documents at a web server and a

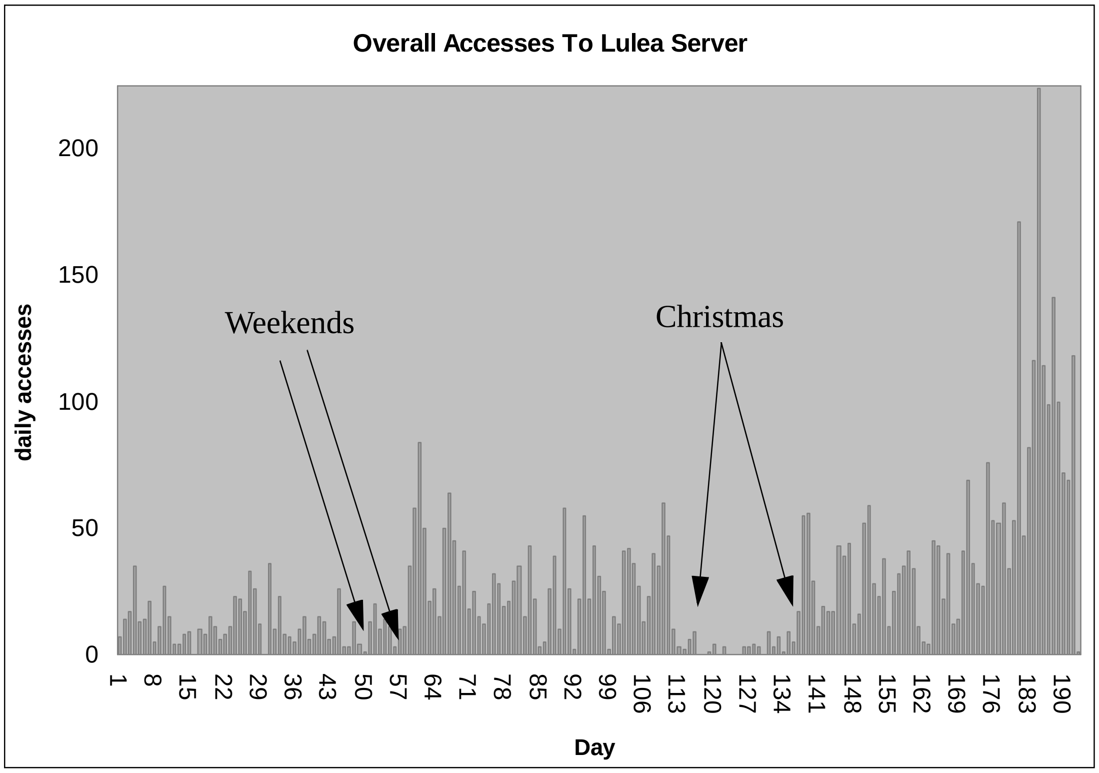


Figure 3.5: Accesses To Video Server

history of access to them, the final ranking of document popularity (*p*) is related to its frequency of access (*P*) by:

$$P \sim 1/(p^{1-t}) \text{ where } t = 0.27 \qquad \textbf{(EQ 1)}$$

This particular version of Zipf's law is based on a study of popularity followed by video store rentals [17]. Equation 1 implies if video accesses follow the Zipf pattern, then a logarithmic plot of video title ranking vs. their total number of accesses should show a straight line. Figure 3.6 indicates this is not the case although there is definitely an access imbalance - the top ten percent ranked titles account for about 50% of all the accesses.

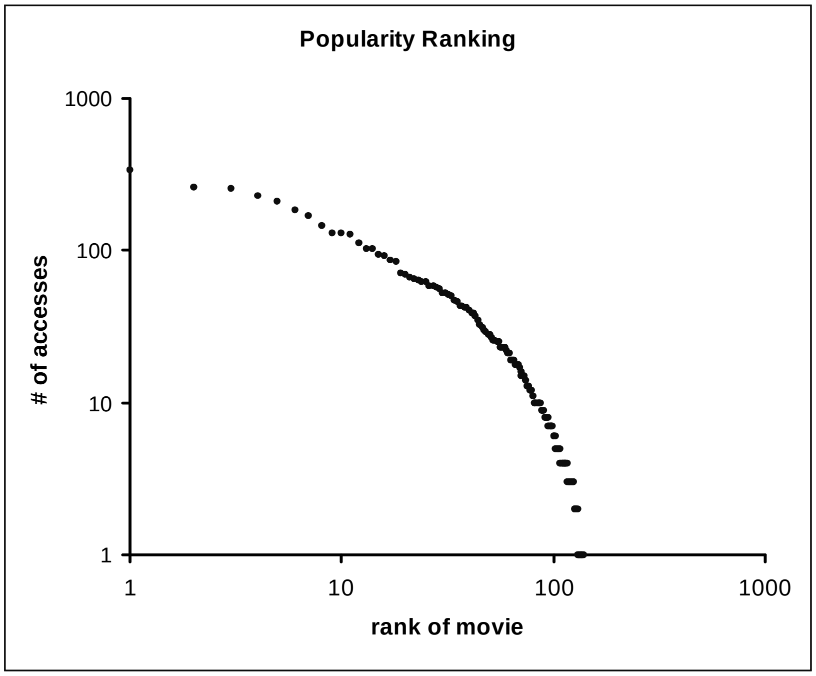


Figure 3.6: Log-Log Plot of Total Movie Accesses vs. Movie

3.4.4 Video Accesses By Machine

The bulk of the machine accesses (67.4%) were local i.e. originated from the campus. Most of these local accesses (63.5%) came from three subnets: *cdt.luth.se* (16.0% of total accesses), *sm.luth.se* (30.1%) and *campus.luth.se* (30.8%). I also found that machine accesses exhibited a skewed pattern: the top ten percent of the requesting machines accounted for about 59% of the total requests.

3.4.5 Inter-access Arrival Times Distribution

In an attempt to detect any patterns in request arrival times, I plotted the distribution of inter-access times of the entire request series. This is shown in figure 3.7. More detail about the distribution of interarrival times less than or equal to 100 seconds is given in figure 3.8. I found the median inter-arrival time

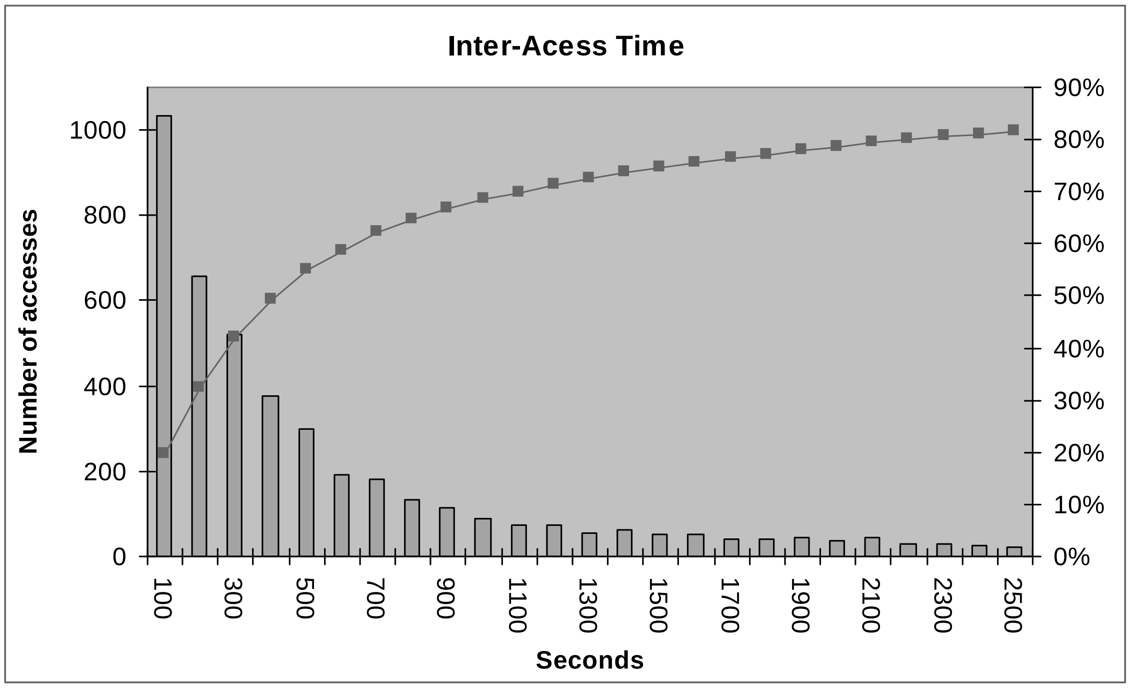


Figure 3.7: Inter-arrival Time Distribution

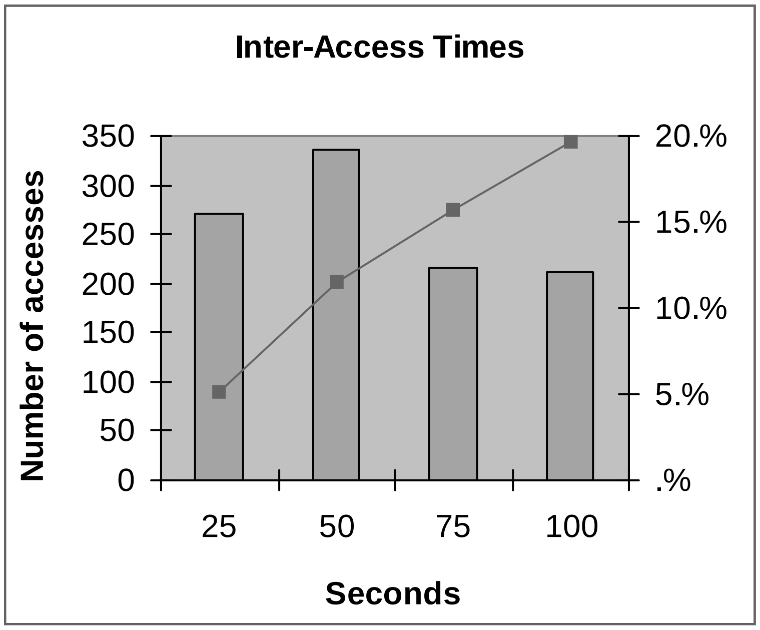


Figure 3.8: Inter-arrival time plot for times $\leq$ 100 secs

to be 411 seconds. With the exception of the observation that time between requests tend to be short, no other clear indications emerged from this plot.

3.4.6 Partial Accesses

Not all of the playback sessions in the trace went all the way to completion. Given the assumption that a user GET request for a title that didn't have a subsequent matching STOP meant the user viewed that title all the way through, I found that about 55% of all requests played the entire duration. Figure 3.9 summarizes the degree of movie playback (before stoppage) for the remaining 45% requests as percentage of movie duration. Most stoppages occurred during the first 5% of the movie playback period.

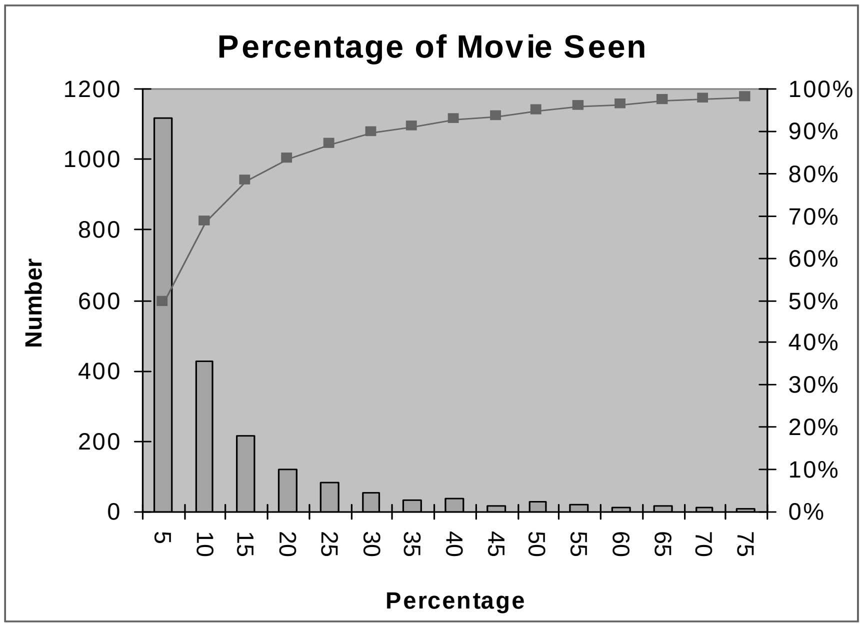


Figure 3.9: Degree of Partial Playback

3.4.7 Access Patterns Vs. Type of Title

The titles hosted by the mMOD server fell into two categories -- *general* and *educational*. The former type was involved sort of entertainment or movie. The latter category included recordings of course-lectures, meetings and seminars. A total of seven titles (5% of the total number of titles) were available in the general section including “Waterworld” and “The Life Of Brian.” Overall, general titles accounted for 12.3% of the total number of accesses. Table 3.1 breaks down the types of videos accessed in the three subtraces.

Table 3.1: Percentage of Accesses Accounted By General Titles

cdt	***campus***	***sm***
14.8%	14.0%	8.2%

Additionally, I found that accesses to general titles tended to be evenly distributed with time, whereas educational clips exhibited very high accesses over a smaller period. For instance, the popularity of material associated with a particular course would be likely to rise on the eve of homework assignments and prelims but die down shortly thereafter. Figure 3.10 provides some examples of this trend. Acesses to “Waterworld” are spread out over the entire time period covered by the trace. In contrast, accesses to “SMD074_980210” and “SMD104_971028,” the former being a recording of a single lecture from a Distributed Multimedia course and the latter, an Object Oriented Programming lecture, show considerable variation over a relatively short period of time.

3.4.8 Temporal Locality Analysis

Temporal locality refers to the notion of the same document being re-referenced frequently within short intervals. I used the standard LRU (Least Recently Used) stack-depth analysis [3] of the trace to measure locality. In LRU stack-depth analysis, when a title is initially referenced, it is placed on top of the LRU stack (ie. position 1), pushing other documents down the stack by one location. When the document is subsequently referenced, its current location in the stack is recorded, and the document is moved back to the top of the stack, pushing other documents down as necessary. After the entire log has been pro-

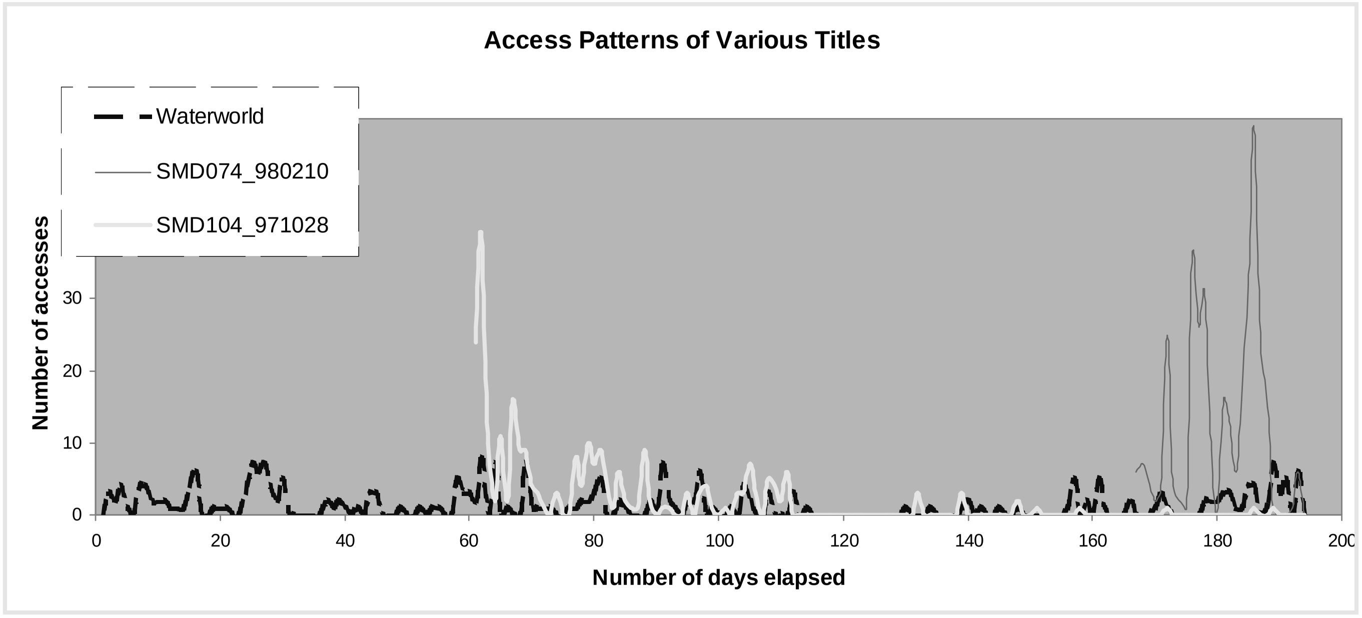


Figure 3.10: Differences in Access Patterns Between the Two Categories

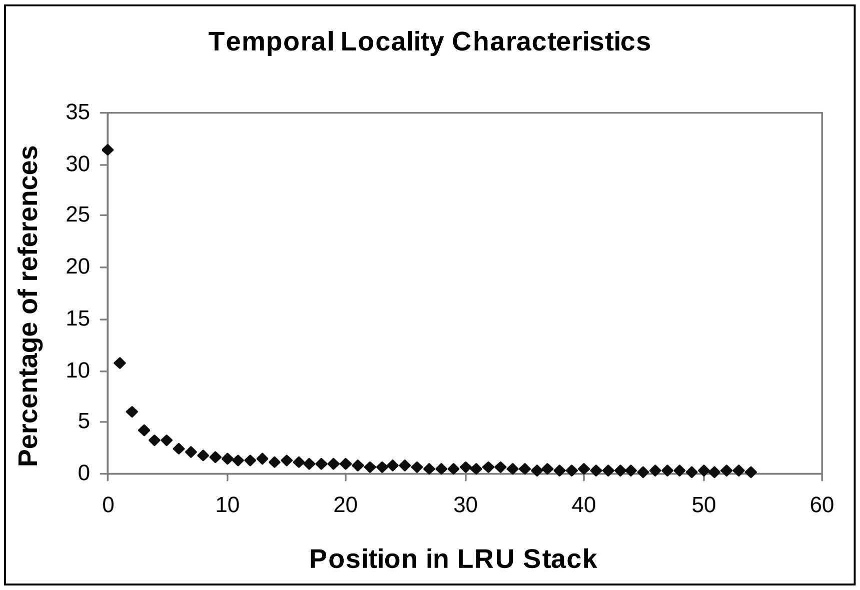


Figure 3.11: Results of LRU Stack-depth Analysis

cessed in this fashion, temporal locality is indicated if the top few positions in the stack account for the bulk of the cumulative references. Figure 3.11 shows the analysis results. The top few positions in the stack account for a majority of overall references, thus indicating that the data does indeed display high temporal behavior.

3.5 General Observations

From the analysis of file characteristics, I found that content creators took advantage of the relatively high bandwidths available in order to create large files with lengthy durations. On the other hand, the trace analysis showed that viewers often accessed movies only partially. This observation, coupled with the high temporal locality present in the trace, indicates some sort of *video browsing* *pattern* whereby a user might click on a title and let it run for a couple of minutes. If interested, the user lets it run to the end or he/she might stop the title, start it again from the beginning and then let it run to conclusion. If not interested, the user simply stops the playback session. Additionally, I found that the category of the movie also affected the type of reference pattern. Access to general titles tended to be even over a long period of time, whereas educational title accesses were more bursty over a shorter time period.

3.6 Related Work

In the absence of any prior surveys of video access over the web, closest related work can be classified into roughly two types: examination of Web traffic and video access analysis for video on demand systems. Web traffic investigations can deal with requests either emanating from a cluster of clients or directly at the server itself. Mogul [39] and Kwan [30] have investigated access patterns at specific servers. In addition to analyzing the underlying systems and network behavior of the server under study, they also examined incoming HTTP requests by looking at their interarrival times, variations with time, size and type of files desired, and requesting domain type. The same core criteria (plus some others) were used by Arlitt [7] to extract underlying patterns from a number of server traces. Cunha et al [18] performed client side traffic work. They instrumented browsers at clusters of workstations to collect individual user access traces, which they then collated and analyzed. In all of these studies, videos accounted for a very small percentage (less than 1%) of overall requests. However, since the traffic data in these studies were all collected during 1994 and 1995 when the web presence of videos was insignificant, they do not present an accurate picture of current video activity.

Most recent video on demand models rely on results reported in two studies: Chervenak [17] and Dan, Sitaram and Shahabuddin [19]. These analyses examined statistics in magazines for video rentals and reports from video store owners. Both studies concluded that the popularity distribution of video titles could be fitted to a Zipfian distribution.

3.6.1 Acknowledgements

I would like to thank Peter Parnes of Luleå University of Technology for graciously providing the traces from the mMOD project.

Chapter 4

The Design of MiddleMan

This chapter presents the architecture of MiddleMan, a video caching proxy server, as well as the concepts and assumptions behind its design. The goal of MiddleMan is to cache videos close to clients and to intercept and service video requests, thus reducing dependence on Internet latencies and web server load. MiddleMan differs from existing proxy research in that it concentrates exclusively on video. Other approaches are optimized for dealing with HTML documents and images. These media possess different properties than video. Hence, these other approaches are unlikely to cache video data effectively.

The design of MiddleMan can be divided into three parts: *system components*, *policies*, and *communication protocols*. System components are the building blocks of the system and their configurations. Policies dictate system behavior such as cache replacement and response to failures in system components. Communication protocols provide the means by which system components communicate.

A thorough understanding of VOW access patterns and file characteristics is an essential first step prior to designing the system. In the absence of any such publicly available work, much of the underlying assumptions behind MiddleMan are drawn from the studies reported in the previous two chapters of this thesis. The next section summarizes these findings.

4.1 Trace Observations

Relevant observations from chapter 2 include:

1. *Web video size:* Videos are around 1 Mbytes in size, an order of magnitude larger than HTML documents which are usually sized around 1-2 Kbytes. Playback time is about a minute or less.
2. *WORM nature:* Web videos tend to follow the write-once-read-many principle. Once a video has been placed online, chances are that it will stay there. Hence, cache consistency is not a major issue in video caching systems. Additionally, snapshots of the mMOD web server file system (discussed in Chapter 3) at various time periods during the trace indicate that once a video file is placed online, it tends to remain online.
3. *High bandwidth requirements:* A high percentage of web video material cannot be downloaded and played back in real-time as current network/modem bandwidths are not enough to meet their implicit playback requirements.
4. *Growth*: the number of videos on the web are growing at an almost exponential rate.

Chapter 3 yielded the following insights:

5. *Future trends:* videos are becoming larger as more network bandwidth becomes available and low bitrate streaming protocols get deployed in video distribution. With a high bandwidth network and H.261 based multicast architecture in place, the median size of files at the Lulea University video server was 110 MBytes. Median duration was 77 minutes.
6. *Inter-arrival times*: the median interarrival time of requests for videos was about 400 seconds. This indicates requests are infrequent compared to HTML documents.

7. *Video browsing patterns:* users like to view the initial part of videos in order to determine if they are interested or not. If they like what they see, they continue watching. Otherwise, they stop.
8. *Temporal Locality:* accesses to videos exhibit strong temporal locality. If a video has been accessed recently, chances are that it will be accessed again soon.

Observations 3, 7 and 8 hint that a caching approach could yield rich dividends. The remaining lessons suggest some basic requirements for the initial blueprint of MiddleMan:

- <u>scalability</u>: the capacity of the caching system must be able to keep pace with the rise in VOW popularity (observation 4). It also must be also be expandable to serve more local clients.
- <u>flexibility</u>: it should be able to handle both files with sizes in the megabyte range (observation 1) as well as the hundreds of megabyte range (observation 5).
- <u>partial files:</u> since users are likely to view only part of the video (observation 7), video files need not be stored in the local caching system in their entirety.
- <u>cache consistency</u> is not a significant issue in the system design (observation 2).

4.2 The Structure of MiddleMan

MiddleMan consists of two types of components: *proxy servers* and *coordinators*. A typical configuration consists of a single coordinator and a number of proxies running within a local area network. A proxy interfaces with users and manages video files. A coordinator keeps track of proxy contents and makes cache replacement decisions for the entire system.

Proxies can run on LAN servers or user machines. Each proxy is responsible for a) responding to client browser requests for video and b) managing a certain amount of local disk space where the video data is cached. Ideally, a proxy would run on every user machine in a domain, but this might be hard to deploy. Hence, I assume proxies run on selected machines in the network. Each proxy services a small collection of browser clients that have been configured to forward their requests for video data to that particular proxy.

Microsoft and Netscape browsers can be configured to forward their requests to proxies via the *client auto proxy configuration system* [89]. According to this system, when a browser is first launched, it automatically downloads a Javascript [90] configuration document from its “home” web server. The document outlines a policy where certain types of browser requests may be redirected to certain proxies while others are left untouched. MiddleMan uses the client auto proxy configuration system to redirect requests for video documents to the proxy server

When a proxy receives a request to fetch a video, it contacts a coordinator. If the title is present in the system, the coordinator returns a list of proxies that store the data. Otherwise, the coordinator selects a collection of dispensable titles currently in the system and asks the querying proxy to fetch the requested title from the web server and replace these dispensable files. Note that a coordinator does not directly handle video data or user requests. Instead, it keeps track of the data managed by each proxy and responds to proxy queries for videos.

A number of issues are raised by this design:

1. How should the proxies be configured or arranged with respect to the coordinators? The proxies could all contact the same coordinator or there could be a hierarchy of proxies and coordinators.
2. How should videos be stored? A video could be stored contiguously on a single proxy or fragmented and saved over multiple proxies.
3. Given our knowledge of video browsing patterns, should the entire video be stored? A possibility is to only save the initial portion of a video locally and fetch the rest from the web server as necessary.
4. How do proxies and coordinators communicate? Variables include the nature of messages exchanges and what underlying protocol (TCP or UDP for instance) should be used to transmit the messages.
5. How should videos be replaced? In other words, what is the most effective cache replacement policy?
6. How to share load across multiple proxies? The design problems include defining a “proxy load” and deciding a cache replacement policy that minimizes proxy load variations.

Section 4.3 of this chapter answers question 1 by further discussing the system component aspects of MiddleMan. I address 2 and 3 by presenting the relevant design policies in section 4.4. Question 4 refers to the system communication protocols, which I detail those in section 4.5. Section 4.6 summarizes related work, and section 4.7 outlines topics, such as questions 5 and 6, that require further investigation.

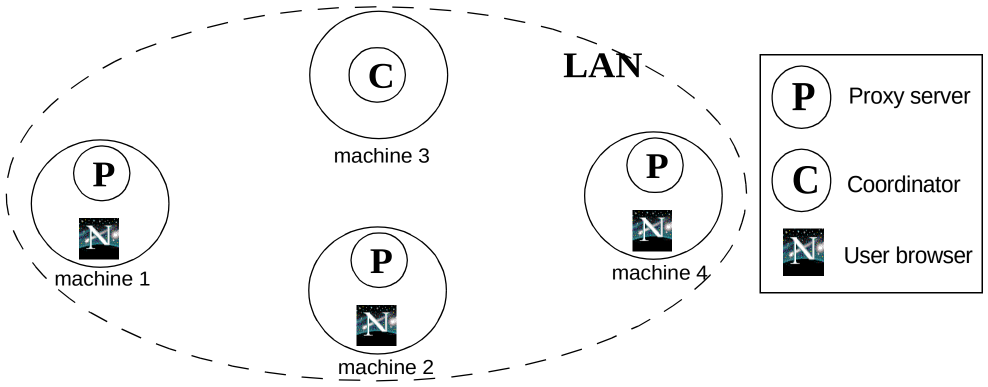


Figure 4.1: Proxy Cluster

4.3 System Component Configuration

The first design question that must be answered is: how are proxies configured with respect to the coordinator? MiddleMan is a collection of proxy servers running on user machines within a local area network. The proxies are organized by a *single* coordinator. Together, they form a *proxy cluster*. Figure 4.1 shows an example cluster consisting of three proxies and a coordinator. Since the proxies consult a coordinator for every request, its central nature might make it a bottleneck. However, the relatively large inter-request arrival times for video, and the fact that the coordinator does not transfer video data, implies this is not a cause for concern.

A proxy cluster based system provides a number of advantages:

- *latency reduction:* communication and data transfers amongst the cluster components can exploit the high bandwidths of the local area network.
- *high aggregate storage space:* by running on user machines, proxies can take advantage of cheap disk space. For instance, if 10 machines within a cluster managed 200 Mbytes each, total space available is 2 Gbytes. Five

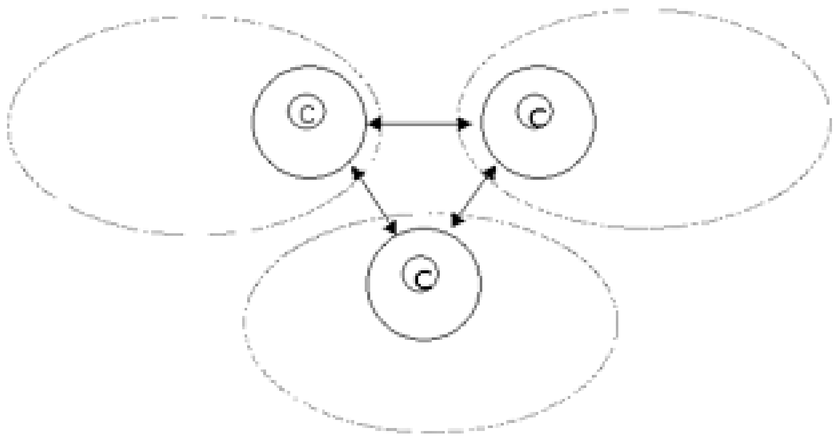


Figure 4.2: Inter-coordinator cooperation

hundred machines on campus could then store about 100 Gbytes worth of movies (all the videos that I located and analyzed in chapter 3).

• *load reduction:* distributing video files on multiple machines allows the load induced by video requests to be distributed over multiple machines. This is better than a central server that services all local video requests and hence becomes a system bottleneck.

• *scalability:* the capacity of the system can be expanded by adding more proxies. Globally, multiple clusters can be linked together by allowing individual coordinators communicate as shown in figure 4.2.

The main disadvantages of this system is that the coordinator is the central point of failure. In case of a coordinator crash, the system loses state. One possible solution is the *coordinator-cohort* approach where the coordinator maintains a backup coordinator. The coordinator keeps the cohort updated with its current state so that, in the event of a crash, the cohort takes over. Since building fault tolerant central servers is a well studied problem, I did not build a fault tolerant coordinator. Nothing in the design of MiddleMan prevents making the coordinator fault tolerant, however.

Proxy crashes are easier to circumvent. The coordinator, upon detecting the failure, updates its own state by invalidating the files managed by the crashed proxy. The coordinator also removes the locks on resources currently held by the proxy.

The distributed nature of the MiddleMan components also complicates video storage since files can be stored in any number of proxies. The next section explores the issues in more detail.

4.4 Video Storage Policies

The next design question is how much of a video should be cached in MiddleMan. I studied two policies: *partial caching* and *full caching*. In partial caching, a video need not be stored in its entirety by MiddleMan. Only a portion of it may be kept by the system. In full caching, MiddleMan deals with the entire video, storing or replacing it entirely as necessary. The next two subsections elaborate on each policy.

4.4.1 Partial Caching

As shown in chapter 3, users are far more likely to view the opening of a movie than play it back until its end. Hence, unlike HTML documents, an entire video document does not have to be present in the caching system for a request to be satisfied. Based on this observation, MiddleMan incorporates the concept of *partial video caching*. When the user requests a video in the cache, it is served by sending to them the portion of the video locally present while obtaining the remainder from the main WWW server and transparently passing it on to the client.

In order for the partial video caching to work, video servers and streaming protocols must allow random access. Fortunately, all major streaming protocols and HTTP 1.1 [91] allow random access.

MiddleMan *fragments* cached videos into equal sized file blocks in the storage system, which allows the cached title to be spread across multiple proxies. The coordinator is configured with the size of a file block. Proxies receive the block size information from the coordinator and use it for reading, storing, and merging pieces of video files.

Blocks at a proxy are addressed via *block slots*. Empty slots at a proxy implies it has space available in multiples of block sizes. A full block slot indicates a portion of a title. The proxy, however, does not keep track of the full/empty status of its own block slots. It also has no knowledge about its block contents. The coordinator fills blocks and determines block content on behalf of each proxy.

Representing video as an ordered sequence of file blocks simplifies the architecture considerably. It provides a convenient mechanism for spreading portions of a single title across multiple proxies, thus allowing for better load balancing, simplification of cache replacement and partial video implementation. If a new title T_1 needs to be brought into the cache, yet the entire system is full, blocks allow MiddleMan to simply eliminate the end portions of an unpopular title T_2 on a block by block basis. Hence, instead of getting rid of T_2 entirely, we can have a portion of it present in case it is requested in the future. Similarly, T_1 grows on a block by block basis, its ultimate size depending on how much of it is played back by the user.

A possible problem with this method of fetching blocks from multiple proxies and combining them into a contiguous stream might be due to *inter-block switching delays.* This can cause interruptions in the data flow from the proxy to the user. Delays may be caused by the latencies faced by the video data receiving proxy when it is changing its source from one proxy to another. Such switching delays can be eliminated via fetching data at a higher rate and double buffering against latencies.

4.4.2 Full Caching

To study the performance effects of partial caching, I analyzed another storage policy, *full caching*. In full caching, a video file is cached in the system in its entirety or not at all. Videos are still fragmented into blocks which can be scattered over multiple proxies, but in contrast to partial caching, these blocks are not independent entities. Instead, if a title is deleted, *all* the blocks associated with it are freed. Similarly, the coordinator allocates space for a new title by reserving all the blocks necessary to hold the entire movie in advance. If sufficient blocks cannot be found, the coordinator does not allow the title to be cached.

4.5 Communication Protocols

In this section I describe the components of the system and their communication needs. I focus on four scenarios: *initialization*, *cache misses*, *cache hits,* and *request cancellations*, leaving other possibilities such as proxy/cache failures for future work.These four conditions and how both variations of the MiddleMan architecture cope with all of them are detailed in the following subsections. I start with how these four scenarios are handled when MiddleMan adopts the full caching storage policy as described in section 4.2.2.

4.5.1 Full Caching

Table 4.1 details all the message types used by the proxies to communicate with the coordinator in response to the various situations. The initialization scenario is the simplest to outline. Upon startup, the proxy locates the coordinator and sends it an *addProxy* request alerting it to the existence of the proxy and the space it manages. The coordinator then modifies its own data structures to reflect the newly available free space. The proxy can be manually configured with the actual location of the coordinator. Alternatively, the proxy can use the auto proxy configuration system to download the location from the default local web server.

The other three cases are more involved. Figure 4.3 illustrates a typical system that will be used in the remaining three cases. The example system consists of a proxy cluster and *W*, a WWW server external to the LAN. No block are currently cached by the system. The proxy cluster contains a coordinator *C* plus proxies P_1, P_2, and P_3 serving three client browsers B_1, B_2, and B_3, respectively. P_1, P_2, and P_3 all have one empty block slot each, s_1, s_2, and s_3. *W* hosts a movie **M** that can be logically divided into two file blocks, M_1 and M_2. Stored blocks are denoted by *{proxy address, block slot},* where "proxy address" is the IP address of the proxy and "block slot" is the slot of the referenced block. I now present how the system reacts to the three scenarios (cache miss, cache hit, and request cancellation) in the following subsections.

Table 4.1: Full Caching Proxy-Coordinator Communication Synopsis

Proxy Command	**Parameters**	**Why**	**Coordinator action**
addProxy	size of cache locally managed by proxy	a proxy sends this message to register itself with the coordinator	Adds the size to the list of free blocks. Returns *ok* if this was successful
query	movie url	a proxy sends this message to the coordinator after receiving a request from its client to fetch a movie	Returns *no* if this title is not present in the system. Otherwise, it returns a list of blocks corresponding to the movie. The coordinator locks these blocks to prevent deletion. The locks have timeouts in case of a proxy crash or some other mishap.
doneUrl	movie url	a proxy sends this message to the coordinator after it is done reading all the blocks from a *query*	Unlocks the blocks corresponding to this movie.
reqSpace	movie url, space requested	following a cache miss, a proxy requests a certain number of free blocks from the coordinator. The total size of the blocks requested is greater than or equal to the size of the movie.	If sufficient free space is not available, it runs a cache replacement algorithm which deletes entire movies until enough space has been vacated. Returns the list of blocks. On the other hand, if sufficient space cannot be freed, the coordinator simply returns *no*.
doneCreatingUrl	movie url	a proxy notifies the coordinator that it has saved a copy of the movie at the locations specified by the coordinator after the *reqSpace* request	Coordinator marks the movie as being created.
cantWriteUrl	movie url	the proxy cannot complete caching the movie since the client has lost interest and cancelled the movie fetch.	If no other proxies are currently viewing the same movie, it returns *stop* to the proxy and, internally, it returns the blocks allocated to this movie to the list of free blocks. If other proxies are currently viewing the movie, the coordinator returns *cont*.

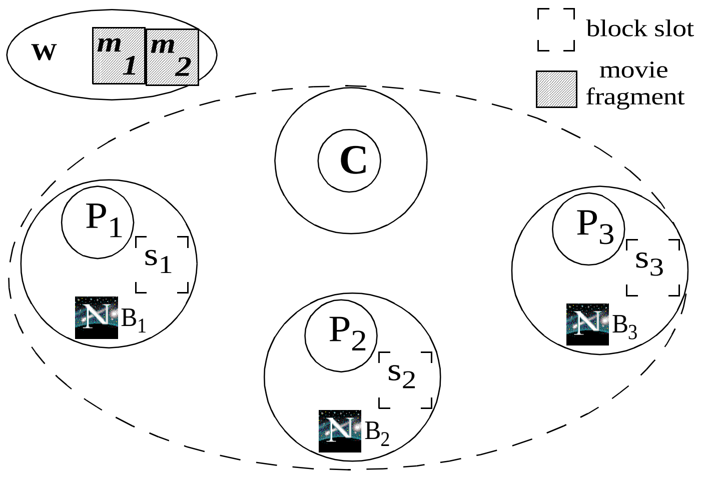


Figure 4.3: Initial state of example system

4.5.1.1 Cache Miss

Figure 4.4 provides a timeline of the events that occur when a browser *B_1* requests a movie **M** that is not cached. When *B_1* requests **M**, proxy *P_1* performs two actions simultaneously. First, it connects to the web server W to request **M**. Second, it contacts coordinator C to ask whether **M** is cached locally. By contacting *W* and *C* simultaneously, *P_1* ensures that MiddleMan will not add any extra time in delivering **M** to *B_1*. Since **M** is not cached, *C* replies in the negative. Meanwhile, the connection to *W* has been established and, from the initial information supplied in the HTTP header, *P_1* determines the size of **M**.

P_1 sends a *reqSpace* message to *C*, to request space where it can cache **M**. If sufficient free space is available *C* replies with a list of blocks (s_2, s_3). *P_1* begins downloading **M** from W which it streams to both *B_1* and s_2 at *P_2*. When s_2 is full, *P_1* streams the remainder of **M** to s_3 at *P_3*. When playback is complete, *P_1* notifies *C* that **M** has been stored. Figure 4.5 shows the state of the system after the entire transaction is complete.

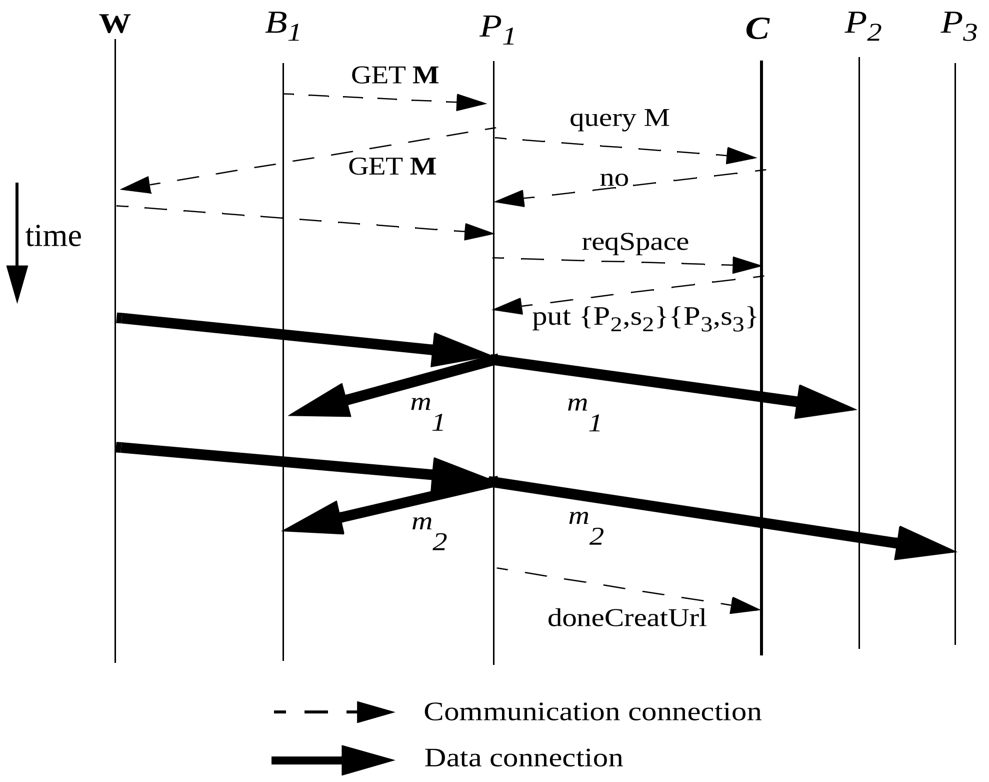


Figure 4.4: Timeline in the event of a cache miss

If the coordinator is unable to allocate sufficient space for **M**, the proxy simply fetches the data from *W* and passes it on *B_1*. Space might not be available for two reasons:

- the size of the title exceeds the capacity of the entire caching system
- too many cached items are currently locked by other users and hence cannot be deleted to free sufficient space.

In the event of coordinator delay in responding to either the *query* or *reqSpace* requests, *P_1* commences fetching **M** and streams it to both *B_1* and a local buffer. After *C* has replied, P_1 saves the buffer in the block slots specified by *C* before storing the portion of **M** currently being streamed to *B_1*.

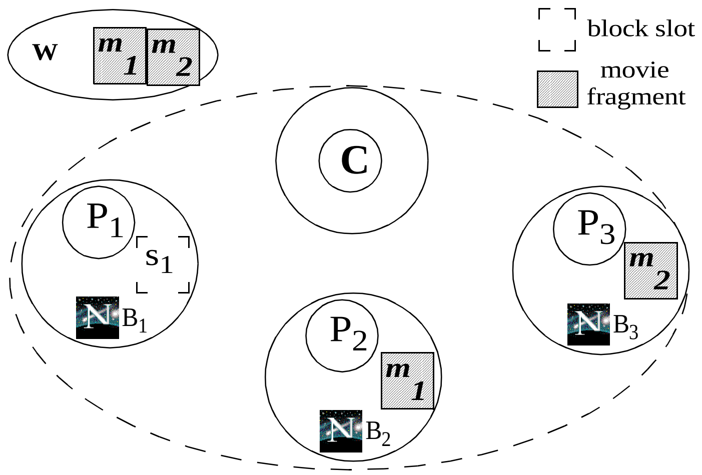


Figure 4.5: Final state of example system

4.5.1.2 Cache Hit

Assuming the system is in the state shown in figure 4.5, and B_1 requests **M** again, figure 4.6 provides a timeline of the sequence of events that transpire. This time, after P_1 has queried both *W* and *C*, the latter replies that it has a cached copy of **M**. P_1 closes its connection with W and fetches **M** from the local caching system. This involves obtaining blocks m_1 from {P_2, s_2} and m_2 from {P_3, s_3} and seamlessly passing them on as a continuous stream to B_1.

4.5.1.3 Request Cancellation

Broken requests are a side-effect of how users browse video - they commence playback of a title and decide to stop if they do not like what they see. Halting the playback results in the browser cancelling its proxy connection while a cache hit or miss type transaction is taking place.

If the title is being fetched from the local cache via a cache hit, dealing with a cancellation is relatively simple. The proxy, after detecting the closed browser channel, simply shuts down the other connections associated with the

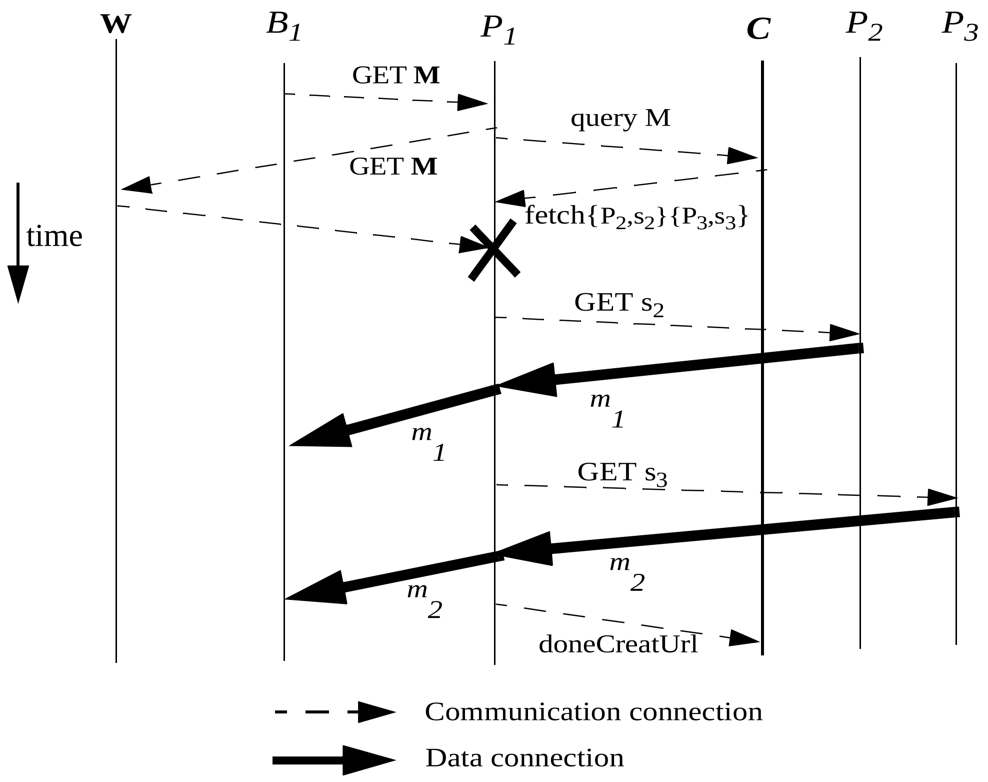


Figure 4.6: Timeline in the event of a cache hit, full caching scenario

request and notifies the coordinator, via a *doneUrl* message, that it is done with the title.

If the request cancellation occurs during a cache miss, complications may arise depending on whether other users in the system are also currently accessing the same file. Figure 4.7 illustrates a cache miss situation with only a single user involved. While viewing the second half of the movie, B_1 decides to cancel its request. P_1 detects this and notifies the coordinator via a *cantWriteUrl* message. As no other users are currently accessing this video, the coordinator asks P_1 to close the remaining connections. Even though block m_1 has already been copied at P_2, the coordinator deletes all references to **M** and adds {P_2,s$_2$}

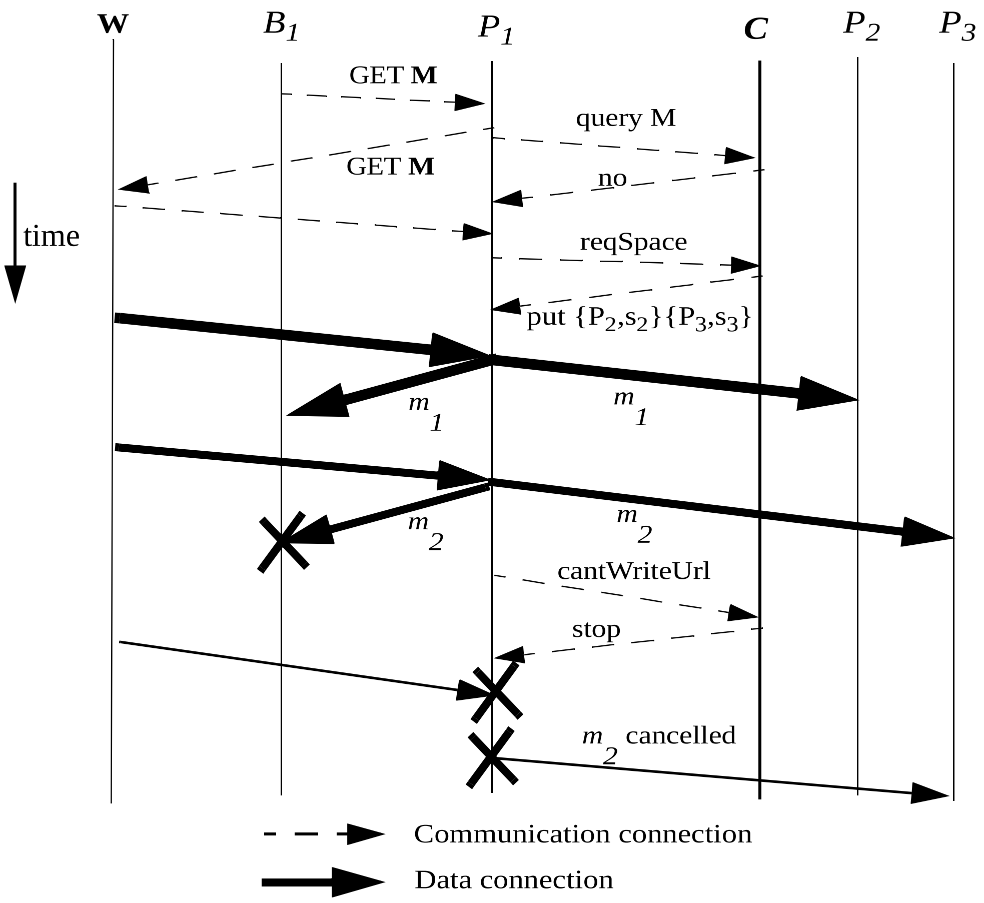


Figure 4.7: Single user request cancellation: full caching

to the list of free blocks. {P_2,s_2} is added to the free list because, under the full caching policy, only entire titles are saved.

Figure 4.8 presents a more complicated scenario where both B_1 and B_2 are concurrently interested in **M** (shown in part (a) of 4.8). B_1 decides to cancel during the second half of the movie playback. Once again, P_1 detects the closed connection and notifies the coordinator. However, since B_2 is still interested, the coordinator asks P_1 to continue fetching the file and save it locally as shown in figure 4.8, part (b). Hence, the final system state is the same as that of figure 4.5.

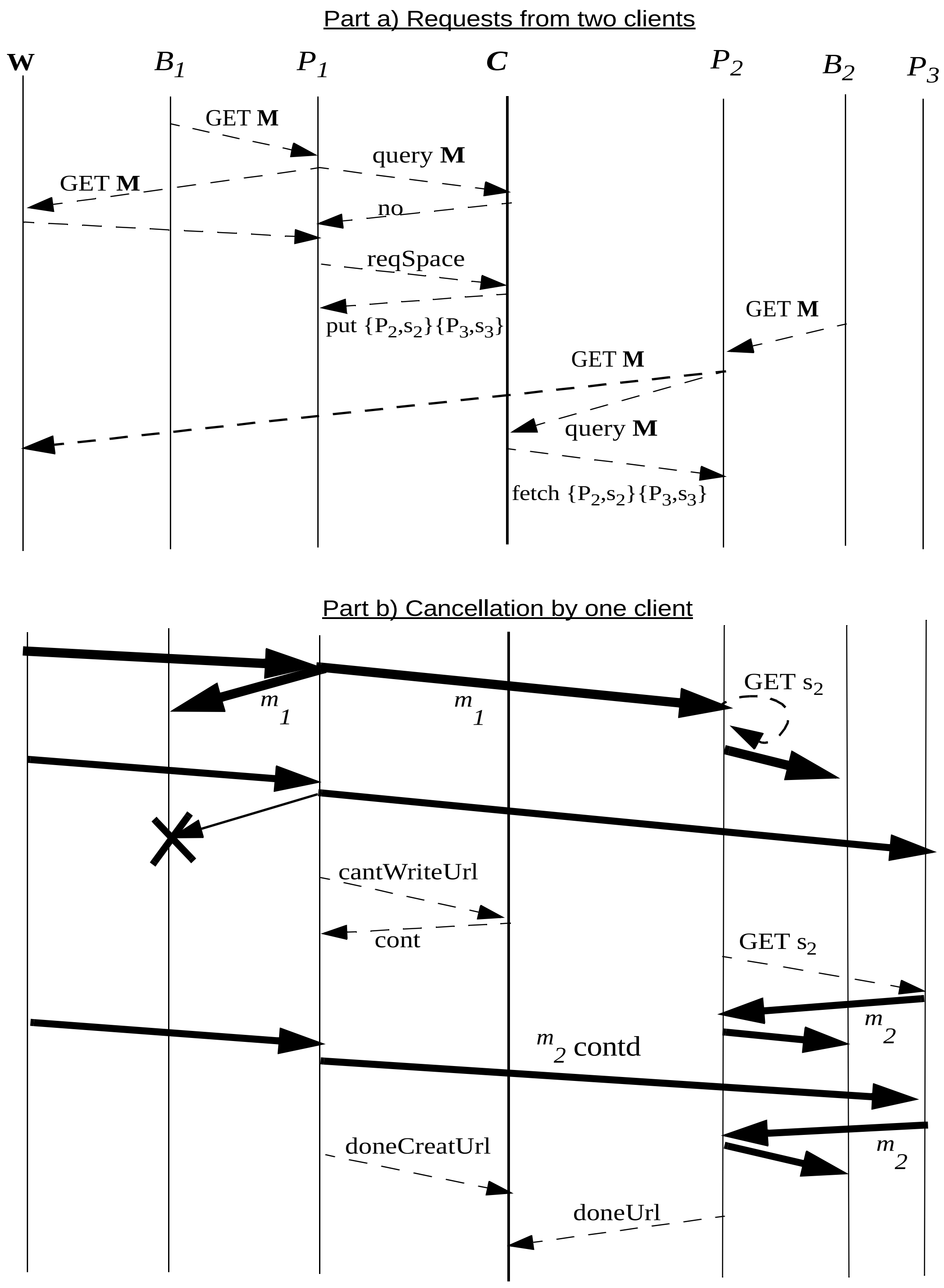


Figure 4.8: Multiple user request cancellation: full caching

A side effect that emerges after the request cancellation in the previous situation is that P_1 bears the additional load of fetching from *W* and streaming to

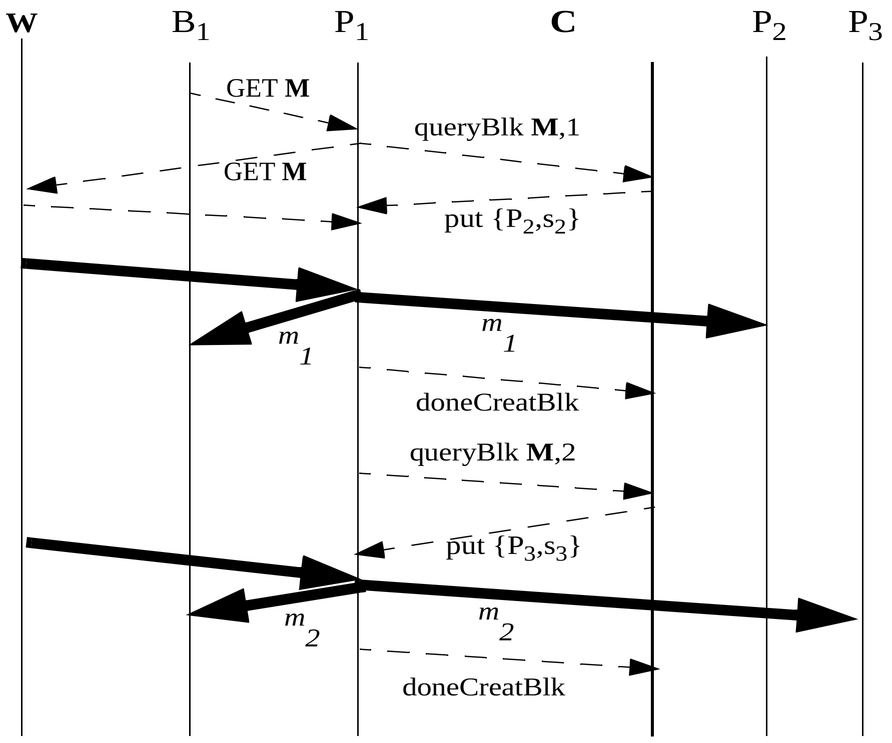


Figure 4.9: Cache miss: partial caching policy

P_2 and *P_3* even though *B_1* is no longer viewing **M**. Handing off the movie fetch from *P_1* to *P_2* would eliminate the burden on *P_1*. However, this approach is not implemented as it would further complicate the protocol.

4.5.2 Partial Caching

We now consider the protocol required by MiddleMan to handle the same four scenarios when utilizing the partial caching storage policy. Table 4.2 outlines the revised set of messages in the protocol. Partial caching necessitates several changes to the original set of MiddleMan protocols. One difference is that under full caching, proxy-coordinator communication only occurred once per movie. Under partial caching, the proxy contacts the coordinator for each block in a movie. The modified exchanges are illustrated in figure 4.9, which reiterates the cache miss scenario, given the same initial system state as in figure 4.3. The system state at the end of this transaction remains the same as in figure 4.5. Similarly, figure 4.10 shows the events occurring in a cache hit in the

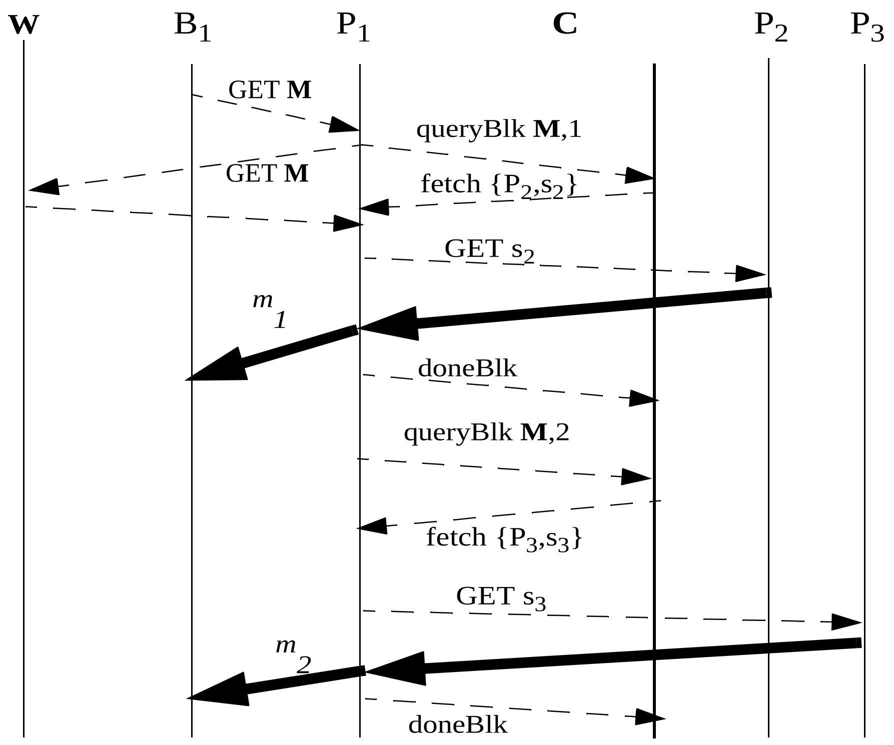


Figure 4.10: Cache hit timeline: partial caching

new system. Note the increase in proxy-coordinator traffic.

Even though the number of proxy-coordinator messages increases, the bandwidth consumed is insignificant compared to that required by the video data transfer. Consider a video file size of 100 Mbytes and a block size of 1 Mbyte. If a proxy-coordinator communication costs 512 byes, the total message transfer cost is about 50 Kbytes or 0.05% of the cost of transferring the video data.

Table 4.2: Partial Video Proxy-Coordinator Communication Synopsis

Command from Proxy	Parameters	Why/Notes	Coordinator action
addProxy	size of cache locally managed by proxy	same as before	same
queryBlk	movie url *u*, block number *k*	a proxy queries the coordinator whether the *k*th constituent block of movie *u* is cached	If block *k* is available locally, it returns the block address. Otherwise, it finds a free block by running the cache replacement algorithm and returns the block address. The coordinator also locks the block
doneBlk	movie url *u*, block number *k*	a proxy sends this to the coordinator after it is done reading a block from a successful *queryBlk*	Unlocks the block.
doneCreatingBlk	movie url *u*, block number *k*	lets the coordinator know it has saved a block after the *queryBlk* request that caused a cache miss.	marks the block as being created.
cantWriteBlk	movie url *u*, block number *k*	the proxy can't complete writing this block since the client has lost interest and cancelled the movie fetch.	If no other proxies are currently viewing the same block, it returns *stop* to the proxy and, internally, it returns the block to the list of free blocks. If other proxies are currently viewing the block, the coordinator returns *cont*.

Note that the design outlined above is still susceptible to coordinator delays. To address this issue, I utilize the double buffer approach outlined in section 4.3.1.1.

Figure 4.11 shows how the single user cancellation scenario is affected by the modified policies. The eventual system state, as given figure 4.12, is different from the outcome of the full caching policy. Under full caching, nothing would be kept locally. However, the final system state under partial caching has block m_1 of **M** cached in {P_2, s_2}. Hence, if **M** is requested again and s_2 has not already been allocated to some other title, the request can still be partially fulfilled by the system. If multiple users concurrently request **M**, the eventual outcome of a request cancellation by B_1 is similar to that of full caching policy i.e. P_1 completes caching **M**.

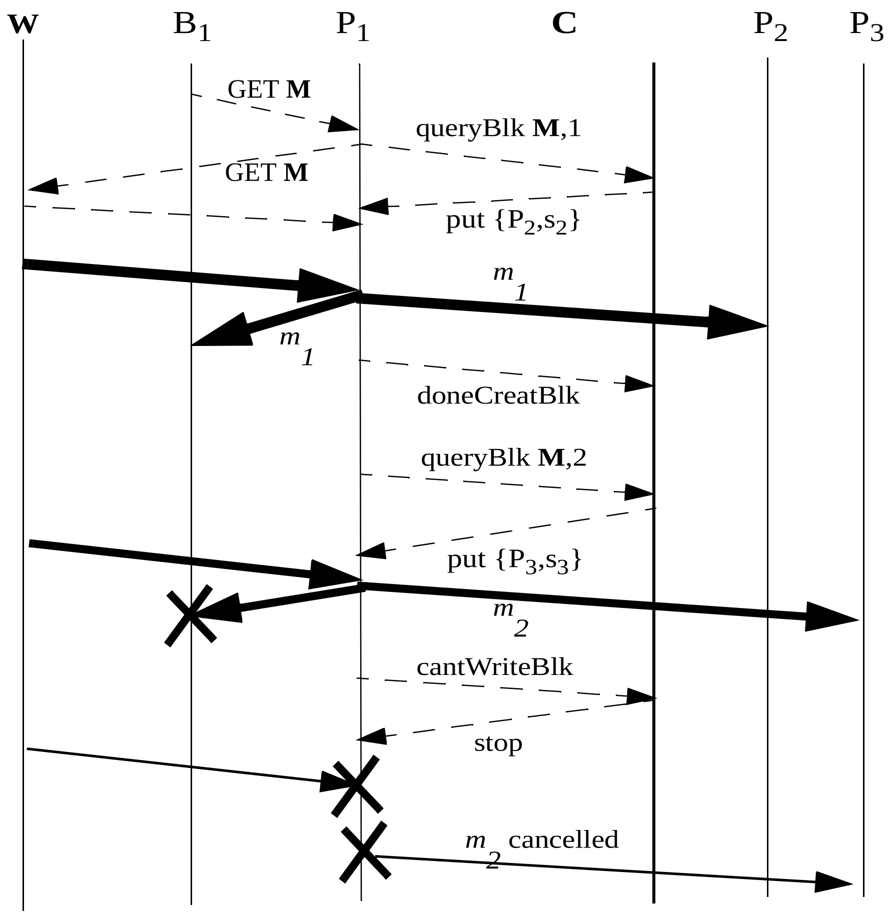


Figure 4.11: Cache miss request cancellation: partial caching

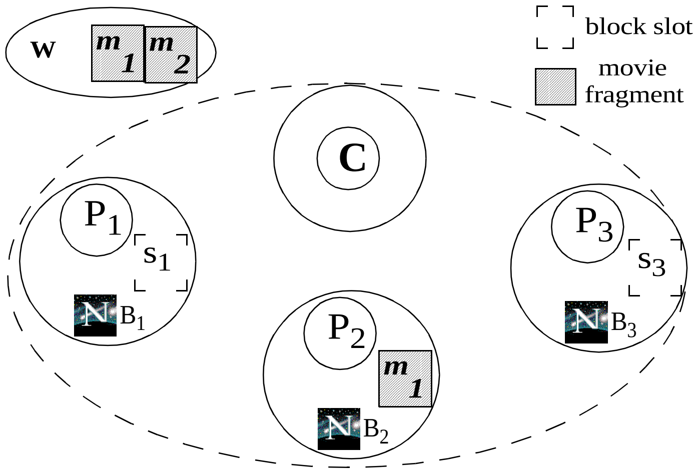


Figure 4.12: Final state after request cancellation

4.6 Related Work

Current research on proxy caching has concentrated on caching HTML documents and images [1, 9, 31]. Little prior work has been done on video proxy caching. Most of the current proxy caching work is not applicable to MiddleMan for the following reasons:

- *Document sizes:* web proxy designs and algorithms are optimized for HTML documents and images, which are generally much smaller than web video files [61, 10]. Video files are undesirable in these systems since if they were to be stored in the cache, they would potentially displace many HTML files. As HTML files are likely to be referenced much more frequently than video files, cache hit rates would suffer.
- *Proxy architectures:* MiddleMan has an unique architecture. The proxy architecture most frequently suggested is a centralized server such as the Netscape proxy server [32]. As discussed in section 4.3, this is not an efficient design for a video proxy server.

Squid or Harvest [16, 92] use a hierarchical design based a tree structure. In these systems, if neighbors of a proxy cannot satisfy a request, the misses propagate upwards through the hierarchy. The miss and propagation combination can add significant latency to final response. Additionally, parents and children caches can potentially store the same files, leading to inefficient use of cache storage space. Hence, such designs have significant drawbacks for caching video.

- *Browsing patterns:* MiddleMan is optimized for video browsing patterns. It supports the notion of partial caching - only portions of video files may be present in the system. Such an approach is not acceptable for HTML documents or images.
- Cache coherence: due to the WORM nature of the vast majority of web videos, this is not a factor for video proxy caches. However, other types of web documents experience high turnover rates and standard proxy caches must deal seriously with cache consistency issues.

The work by Brubeck and Rowe [11] closest to ours. They introduce the concept of multiple video servers that can be accessed via the web. These video servers manage other tertiary storage systems - popular movies are cached on their local disks. They also pioneer the concept of a movie being comprised of media objects scattered over proxy servers. Tewari et al [58] present a resource based caching algorithm for web proxies and servers that is able to handle a variety of object types based on their size and bandwidth requirements. However, they still assume a central non-cooperating architecture.

4.7 Further Work

The design of MiddleMan can be divided into three parts: system components, policies, and communication protocols. This chapter has presented the basic system components and communication protocols. The remaining information, namely the component parameters (e.g., cache size and block size) and the system policies (caching algorithms and load balancing) are studied in the next chapter.

Chapter 5

The Analysis of MiddleMan

In this chapter, I investigate the performance of MiddleMan via extensive simulations. In particular, I look at the performance effects, both in terms of cache hit rates and load balancing among proxies, of varying the cache replacement algorithm, the global cache size, and the number of proxies in the system.

I selected the simulation approach for evaluating MiddleMan because it offers several advantages over theoretical techniques such as queueing models. These analytical approaches have difficulty modeling dynamic behavior such as user request cancellations. Additionally, as chapter 3 illustrates, accesses to video titles and inter-request arrival times do not follow the assumptions common in probabilistic analyses of video accesses. Moreover, the sheer number of individual components within the MiddleMan architecture such as proxies and file blocks, and their distributed locations, further complicate a theoretical analysis. Hence, simulations offer a better method of studying the architecture and provide insights into the dynamics of protocols that queuing analysis does not.

With the exception of section 1, which describes MiddleSim, the simulator used to analyze MiddleMan, the remainder of this chapter is structured as a series of analyses into the various properties of the MiddleMan architecture. Section 5.2 explores cache replacement policies. Section 5.4 investigates load balancing schemes for MiddleMan. Section 5.5 discusses the performance effects of altering the block size. I summarize overall findings in section 5.5.

5.1 MiddleSim

MiddleSim is the simulator I developed to evaluate the performance of cache replacement policies and load balancing schemes in the MiddleMan architecture. The final version of the simulator was about 5000 lines of Java code. The design of MiddleSim went through several iterations. I provide a brief history of this evolution in section 5.2.1, outline the main MiddleSim assumptions in section 5.2.2, describe the software architecture in section 5.2.3, sketch the inputs and outputs of the simulator in section 5.2.4, and finally, present the techniques I used for verifying MiddleSim in section 5.2.5.

5.1.1 History

Initially, I built a full-caching prototype of MiddleMan in order to demonstrate proof of concept and achieve a better understanding of the design space. The prototype was built as an extension of Jigsaw [93], a freely available web server written entirely in Java [94]. The coordinator was a separate Java program. The experiences gained from the prototype led to the creation of a simulation environment, MiddleSim, that could be used to investigate various MiddleMan scenarios. The current version of MiddleSim is written entirely in plain Java. In the next section, I describe the assumptions integrated into this version of MiddleSim.

5.1.2 Assumptions

Picking the right set of assumptions for a simulator can be a tricky affair. Wrong assumptions can lead to invalid results. Too few assumptions can yield unusable simulation results, while too many complicates implementation and belies the ultimate purpose of a simulator, which is to explore design trade-offs. Bearing this in mind, MiddleSim models an environment where network conditions are simplified and modes of failures restricted.

5.1.2.1 Network/Message Passing

MiddleSim makes no specific assumptions about the underlying network or transport protocols used to transmit video, merely that video is streamed at a steady rate. In essence, MiddleSim simulates the timely flow of video data from one MiddleMan component to another. Additionally, MiddleSim deems the cost of non-video communications, such as those between the proxies and the coordinator, to be insignificant compared to the cost of video communication. For example, assuming a block size of 1 Mbytes and a file rate of 100 Kbytes/sec, a proxy only needs to contact the coordinator every 10 seconds. Assuming a proxy-coordinator message transaction size of 100 bytes, the proxy will total 200 bytes for messages processed before and after a video block. In other words, for every block the proxy-coordinator communication costs account for 0.02% of all bytes processed by a proxy. Hence, given the small ratio of non-video to video communication costs, I constrain the proxy-side non-video communication costs to be insignificant. The coordinator, on the other hand, solely processes protocol related messages from proxies. Given that processing a single message essentially involves coordinator data structure manipulation, I assume its cost to be negligible. I also ignore the aggregate costs of serving multiple messages for the following reasons:

- a proxy only generates coordinator messages at relatively large intervals
- trace analysis shows that only a small number of proxies are active i.e. serving video streams at any given time

Hence, the total number of requests processed per second by the coordinator is likely to be small. Thus, I ignore their aggregate processing costs.

5.1.2.2 Failure Model

MiddleSim assumes reliable network connections and machines. The only failure simulated is that of clients cancelling their requests for videos.

5.1.3 Architecture

To comprehend the MiddleSim architecture one must understand the software structure of the simulator, its inputs and outputs, its run-time behavior, how it is verified, and finally, the simulation parameters. These are described in the following subsections.

5.1.3.1 Software Architecture

MiddleSim is composed of a number of modules which approximate the functionality of their real-life counterparts. Figure 5.1 illustrates all the components and how they integrate into MiddleSim.

- The *BasicServer* object emulates a web server via indexing and serving a list of files.
- The *ProxyServer* object can both serve files to, and receive files from, other ProxyServers.

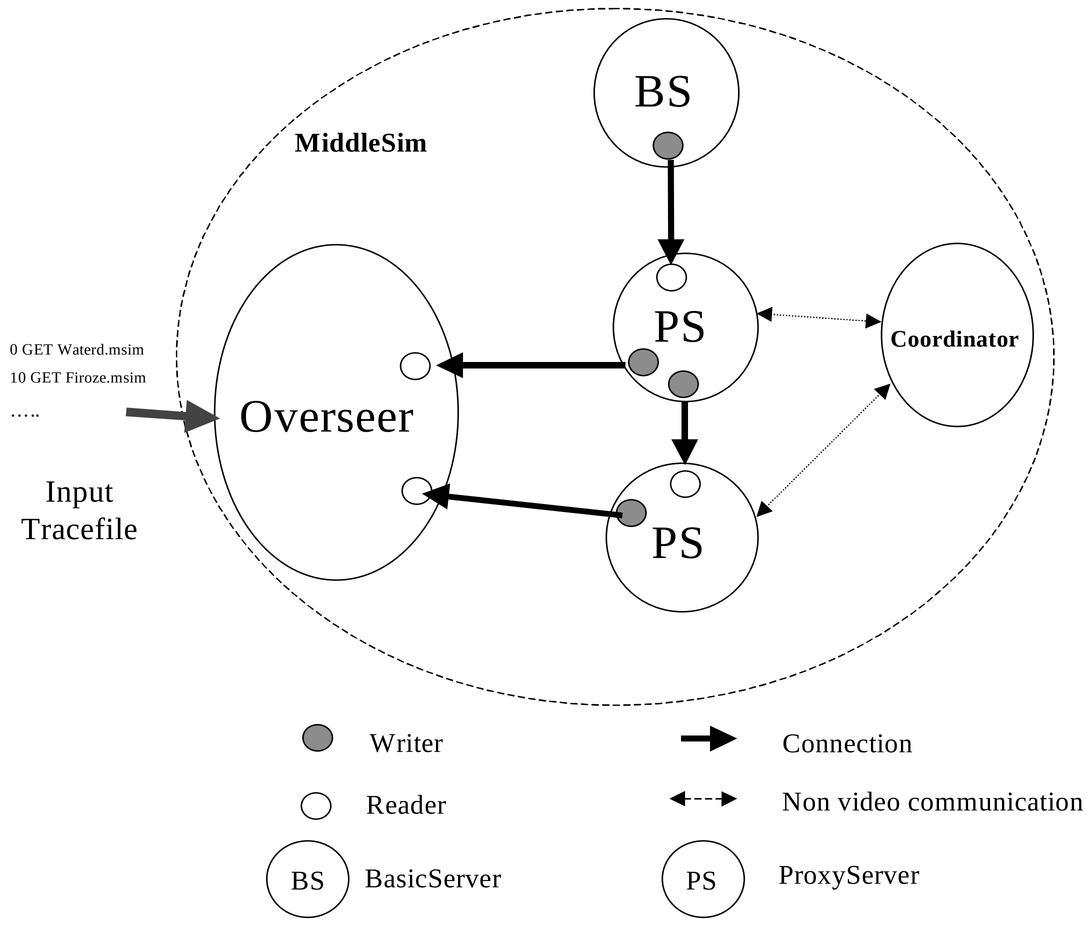


Figure 5.1: MiddleSim Architecture

- The *Coordinator* object keeps track of the state of all proxy servers and ongoing data transactions, allocates blocks on proxy servers, and runs the cache replacement algorithm as appropriate.
- The *Overseer* module manages user requests, directing them to the appropriate proxy servers as necessary. It also manages the central event loop of the simulator.

Communication between the modules is accomplished via two mechanisms, depending on whether the exchange involves transfer of information or video data. All control communications are simulated via procedure calls from one module to another and hence occur instantaneously, per MiddleSim assumptions. Video data is transferred via *Connection* objects, which simulate and track data flow. Reading and writing data to and from the connection is simulated by *Reader* and *Writer* objects. These objects track how many bytes to transfer per second over the connection, the size of the data remaining, and when to close the connection. Reader and Writer objects are used by higher order objects, such as the BasicServer, ProxyServer and Overseer, to manage their connections.

In MiddleMan, a block is the basic unit for data transfer between components. This policy is implemented in MiddleSim via the exchange of *FileBlock* objects. A connection is responsible for transferring a block of data between the reader and the writer. The block transfer rate depends on the original rate of the video file that the block represents.

MiddleMan also allows two different video storage policies, full and partial, that determine how the component blocks of a video file are treated in the event of a fetch, miss, or request cancellation. Testing both policies involved creating two different versions of MiddleSim. The core modules remained the same in both cases. However, the control protocols were different.

5.1.3.2 Inputs and Outputs

MiddleSim is a discrete-event trace driven [29] simulator. In particular, the OverSeer uses traces from the mMOD project [44], as described in Chapter 3, in order to trigger user request and cancellation events. The other input to MiddleSim is a text file containing a list of titles that are accessed in the input trace, including their sizes and durations. This list is used to initialize the BasicServer object in MiddleSim. The file characteristics are used to calculate the mean transfer rate of that file. The mean rate then determines the speed of data transfer in all transactions involving that particular file.

During a trace execution, all data transfers between components via connections are logged. After a run, the simulator outputs the number of bytes accessed from the Web server, the total number of bytes transacted in the system, the total number of bytes exchanged between the proxies, the bytes served by each proxy, and other statistical data. Of particular interest is the byte hit rate, which is the percentage of the ratio of the bytes accessed locally to the total data accessed by all the users in the system, and the proxy load information, which indicates system balance.

5.1.4 Runtime Behavior

After the web server and all the proxies have been initialized, the Overseer orchestrates the runtime behavior of the simulator by entering an event loop. Loop events may arrive from the queue of trace events, from the list of current connections managed by the proxies, the web server, or the Overseer itself.

The minimum unit of time within the simulator is one second. All non-data communications are assumed to occur within this timespan. However, only one seconds worth of video data is transferred over a connection during this time. The simulator executes until all trace events have been exhausted and there are no current connections remaining.

The pseudocode for the event loop is as follows:

```
while (true) {
       topEvent = traceQ.peek ();
       if (topEvent!= nil) {
              if (topEvent.triggerTime == currentTime) {
                     // do nothing
              } else {
                     if no other events on any queue (i.e. simulator is idle)
                            advance system clock to next trace event
              }

              topEvent = traceQ.deq ();
              if (topEvent.eventType == GET)
                     launch a local connection to a proxy on behalf of a user
              else if (topEvent.eventType == STOP)
                     stop a local connection
       } else {
              // no more events on the queue
              if no other events on any queue i.e. simulator is idle
                     print summary statistics
                     exit simulator
              }

        for all the proxies p
               process all the current connections on p

      process all our current connections
      update stats for all the proxies
      advance current time by one second
}
```

5.1.5 Verifying MiddleSim

I verified the functioning of MiddleSim via two approaches, *null tests* and *sample traces*. Null tests are simple worst case scenarios designed to test basic proxy caching behavior. One such test involves the generation of video requests in strict round robin fashion and varying the size of the cache. If the cache capacity is less than the total size of the files at the main WWW server, a round robin sequence of cache requests will generate no hits. For instance, consider a WWW server that serves three files, m_1, m_2, and m_3, and a video cache that has a total capacity less than the size of m_1, m_2, and m_3. If the cache receives a series of requests in the sequence {r_1, r_2, r_3, r_1, r_2, r_3...} where request r_1 is intended for m_1, r_2 is for m_2, and so on, then no file is kept long enough in the cache for a hit to occur. Now, if the capacity of the cache is increased to greater than or equal to the size of m_1, m_2, and m_3, then the cache is able to store all the files and the hit rate rises to the theoretical maximum.

In addition to the null tests, I also generated sample traces with known results in order to manually verify the functioning of MiddleSim. Finally, I incorporated a number of sanity checks into the simulator responsible for tracking conditions such as the number of bytes being transferred over a connection. If these checkpoints detected any anomalous behavior, they would either output an error message or halt the simulation.

5.1.6 Simulator Variables

A number of MiddleSim parameters can be varied prior to starting a simulation run. These include:

- the cache size of each proxy,
- the size of a file block,
- the mapping - a number of users can be mapped to the same proxy throughout the run. Mapping allows us to maintain the same number of proxies across multiple traces each of whom has a different number of machines generating requests, and
- the cache replacement policy

In the subsequent sections, I investigate the effects of varying these parameters on the simulator output, focusing particularly on the cache replacement policies.

5.2 Cache Replacement Algorithm Analysis

My first set of experiments focused on the effectiveness of various replacement algorithms. I use the *byte hit rate* (*BHR*) metric to evaluate the performance of the various approaches. The byte hit rate (BHR) of a simulation run is defined as:

BHR = *(total bytes served from the cache)/(total bytes read by all clients)* **(EQ 2)**

A BHR close to 1 implies good performance since most of the bytes requested by the users are served from the local cache.

To run this experiment, the other simulator parameters must be constrained to reasonable values. I describe these values before providing more detail on the caching algorithms.

5.2.1 Mapping

In order to effectively compare between the three traces from Chapter 3, I set the total number of proxies in the system to 44 for each simulator run, regardless of input trace. For the first trace, *cdt*, this involved a simple one-to-one mapping of 44 user machines to 44 proxies such that $machine_0$ always sends its request to $proxy_0$, $machine_1$ to $proxy_1$ and so on. The other two traces, *campus* and *sm*, required 97 and 110 machines respectively, to direct their requests to 44 proxies. Here, I constrained the mapping so that in addition to $machine_0$ always placing its request with $proxy_0$, $machine_1$ to $proxy_1$, $machine_{44}$ would contact $proxy_0$, $machine_{45}$ to $proxy_1$ and so on.

5.2.2 File Block Size

I fixed the file block size at 1 Mbyte. This value allows MiddleMan to easily store both the relatively small web video files analyzed in chapter 3 and the much larger video recordings analyzed in chapter 4. Files of the latter type have much lower bitrates than web video (median bitrate of 150 kbps as opposed to 700 kbps for web video). Hence, in the partial storage case, a file block from a video with bitrate of 150 kbps is likely to provide a proxy with a gap of about a minute before the proxy has to contact the Coordinator again. Such a long interval reduces the load on the Coordinator, especially if it is involved with a number of simultaneous connections.

5.2.3 Proxy Cache Size

The aggregate size of all the files stored by the WWW server is about 15.7 GBytes. The total size of the proxy caches can be a small fraction of this total. I selected three proxy cache size configurations: the first allocated 12 Mbytes of cache space to each proxy for a total of 44*12 or 528 Mbytes of global cache storage (about 3.3% of the total size of all the video files). The second configuration allowed 25 Mbytes * 44 or a global cache size of 1.07 GBytes (about 6.8% of total file size) and the third configuration provided 50 Mbytes per proxy for a total of 2.14 GBytes or 13.7% of total file size.

5.2.4 Cache Replacement Policies

I initially investigated three basic cache replacement policies -- *Least Recently Used* (LRU) [53], *Least Frequently Used* (LFU) [53], and *First In First Out* (FIFO) [53]. To compare the effectiveness of these algorithms, I implemented two other approaches, *Perfect* and *Infinite*. “Perfect” implements an ideal cache replacement mechanism [53] that uses knowledge of the future to replace the cache block that will not be used for the longest time. It can be shown that, given a finite cache size, this algorithm is optimal. “Infinite” assumes a cache configuration with space greater than the aggregate size of all the video files. Running the simulator with this configuration yields the greatest cache hit rate possible, regardless of the cache replacement algorithm used since no data is ever removed from the cache.

In addition to the basic algorithms, I examined other replacement policies. Of these, I report on *LRU-k* and *histLRUpick*. LRU-k [41] maintains a history of the previous *k* accesses to each title in the cache. The *k-distance* of a title at a certain time is defined as the difference between the current time and the time at which the *kth* access was made to that title. LRU-k chooses to replace the title with the largest k-distance. It resolves ties by picking the title which has been referenced least recently, i.e., running the LRU algorithm on the tied candidates. LRU itself is a special case of LRU-k where k is 1.

HistLRUpick runs LRU-2, LRU-3, and LRU-4. Ties are resolved by picking the block that is managed by the least loaded proxy. The criteria for choosing the least loaded proxy is presented in section 5.3 (the *HistLoad* metric).

5.2.5 Results and Discussion

Table 5.1 summarizes the results of running the simulations in both the full and partial cases. The following trends emerge from the investigation:

- The hit rate in the partial caching is always higher than in full caching regardless of trace or replacement policy. In the full storage policy, a file being loaded into the cache will be completely deleted if the request is cancelled.

Table 5.1: Byte Hit Rates Under Various System Configurations

Trace	***cdt***		***campus***		***sm***	
Caching Pol-icy	*Partial*	*Full*	*Partial*	*Full*	*Partial*	*Full*
Configuration	***44 machines * 12 Mbytes (3.39%)***					
Perfect	48.78%	34.80%	53.01%	35.46%	61.82%	53.45%
LRU	42.95%	31.44%	49.64%	33.64%	57.76%	50.80%
LFU	43.84%	32.28%	49.03%	33.96%	57.19%	50.88%
FIFO	39.90%	31.19%	47.56%	33.39%	55.34%	50.80%
LRU-3	44.89%	31.71%	49.34%	33.79%	58.74%	51.62%
histLRUpick	44.58%		50.07%		58.85%	
Configuration	***44 machines * 25 Mbytes (6.8%)***					
Perfect	67.64%	56.38%	70.99%	56.98%	75.07%	68.56%
LRU	55.15%	49.86%	61.88%	52.01%	63.57%	61.97%
LFU	59.82%	51.97%	63.90%	50.89%	64.44%	63.02%
FIFO	50.38%	48.90%	57.89%	48.92%	63.97%	62.27%
LRU-3	59.59%	52.01%	64.30%	52.26%	70.13%	64.16%
histLRUpick	60.10%		64.77%		70.05%	
Configuration	***44 machines * 50 Mbytes (13.7%)***					
Perfect	81.44%	78.76%	86.19%	83.04%	86.17%	84.64%
LRU	65.33%	61.53%	76.33%	66.61%	74.15%	71.44%
LFU	73.79%	66.34%	75.47%	71.07%	76.27%	74.05%
FIFO	64.87%	58.82%	69.85%	60.91%	67.86%	70.16%
LRU-3	76.67%	72.93%	81.98%	77.14%	80.97%	77.30%
histLRUpick	76.05%		82.61%		81.40%	
Configuration	***Infinite***					
Infinite	86.83%	85.71%	91.90%	90.98%	92.52%	92.03%

Hence, if the file is requested again, it will have to be re-loaded into the cache. In the partial approach, the portion of the file read into the cache prior to the cancellation (up to the file block boundary) would be left untouched, resulting in a cache hit if the file is requested again.

- Byte hit rates in the *sm* trace are slightly higher than their *cdt* or *campus* counterparts. Further investigation of the *sm* trace showed that it had higher temporal locality than the other two traces, accounting for the better performance.

- The difference between full and partial hit rates is more pronounced with lower global cache sizes because the full policy becomes ineffective when the video file size is greater than available cache space. For example, in the 44*12 scenario, video files larger than 528 Mbytes cannot be cached at all using the full policy. In such cases, a portion of the video file can be cached under the partial policy, resulting in higher hit rates.
- Larger global cache sizes increase the overall hit rate. As the “perfect” run for the 44*50M scenario indicates, it is possible to approach the maximum hit rate while employing a global cache size that is only 13.7% of the total file size.
- The basic cache replacement algorithms do not perform well when cache sizes grow larger. However, when the global cache space is small, the difference between the replacement policies and “perfect” is less pronounced. This indicates overall cache size is more of a barrier to performance than replacement policies when the cache size is small. As cache sizes grow, LFU generally provides the best performance, but there is still a sizeable gap between LFU and “perfect,” suggesting further room for improvement.
- LRU-k[1] improves byte hit rates significantly over LRU at large cache sizes. The better results can be attributed to the ability of LRU-k to exploit temporal locality better than the other caching policies. Examination of the traces revealed that user requests for a particular file tend to arrive at the web server in clusters. By saving the last k references and choosing on basis of the k-th reference, LRU-k ensures that movies which have been referenced multiple times recently are less likely to be removed from the cache since they will

[1] In table 5.1, LRU-3 is explicitly shown but the performances of LRU-2, LRU-3 and LRU-4 are comparable.

have smaller k-distances. Since these same movies will most probably be referenced soon in the future, LRU-k can achieve high hit rates.

5.2.6 Other Algorithms Investigated

In addition to the algorithms presented in this analysis, I also investigated some others, notably LP-alpha and variations on LRU-k. LP-alpha [12] is a hybrid of LRU and LFU. Variations on LRU-k included:

- *BalLRUk* is a variation on LRU-k where ties between candidates for block replacement are broken by picking the block managed by the least loaded proxy.
- *LRUvark* builds on balLRU-k. For a given value of k, LRUvark runs balLRU-k, balLRU-k+1 and balLRU-k+2 to obtain three candidates from which it picks the block managed by the least loaded proxy.

These variations do not perform as consistently well as HistLRUpick, hence, I do not report their results.

5.3 Load Balancing Analysis

Since MiddleMan is a distributed system handling large volumes of data, it is important to examine how well it is able to balance load across multiple proxies. In the previous section, we saw that the BHR of histLRU is within 3-8% of an optimal policy. In this section, I explore the load balancing properties of the various caching algorithm. I only report the results for histLRU and LRU, since LFU, FIFO, LRU-k, balLRUk, LRUvark, and LP-alpha are similar to LRU.

My initial study of load balance among proxies revealed tremendous load disparities regardless of caching policy. Upon reflection, it became clear that in

MiddleMan, the load on a proxy is due to two types of activities: 1) servicing its clients and 2) servicing other proxies.

It is not possible to regulate activities of type 1, since the best way of doing so requires a client with a request to somehow identify the least loaded proxy in the system and contact it, as opposed to the current scheme where the client has a designated proxy which it contacts with all requests. Implementing such a redirection mechanism would drastically complicate the architecture of MiddleMan and increase the delay for servicing user requests. On the other hand, the load due to type 2 activities can be better controlled by using mechanisms that carefully select the proxies where blocks should be replaced or accessed. Since the current MiddleMan architecture does not distinguish between the two load types, some proxies are overused for type 1 activities and thus underutilized for type 2 activities. To address these issues, I made an architectural modification to MiddleMan that I describe in the next section.

5.3.1 MiddleMan Architectural Modification

The key alteration to MiddleMan involves the creation of two different types of proxies: *local* and *storage* proxies. Local proxies run on the same machine and are responsible for answering client requests. They do not store any data and are functionally similar to browser plug-ins. Storage proxies do not directly service client requests, they just store data blocks. Storage proxies can be located anywhere on the local area network. The MiddleMan communication protocol remains the same -- the coordinator merely respects the type of each of its proxies in making decisions. Figure 5.2 shows how the new components of MiddleMan might look in one configuration.

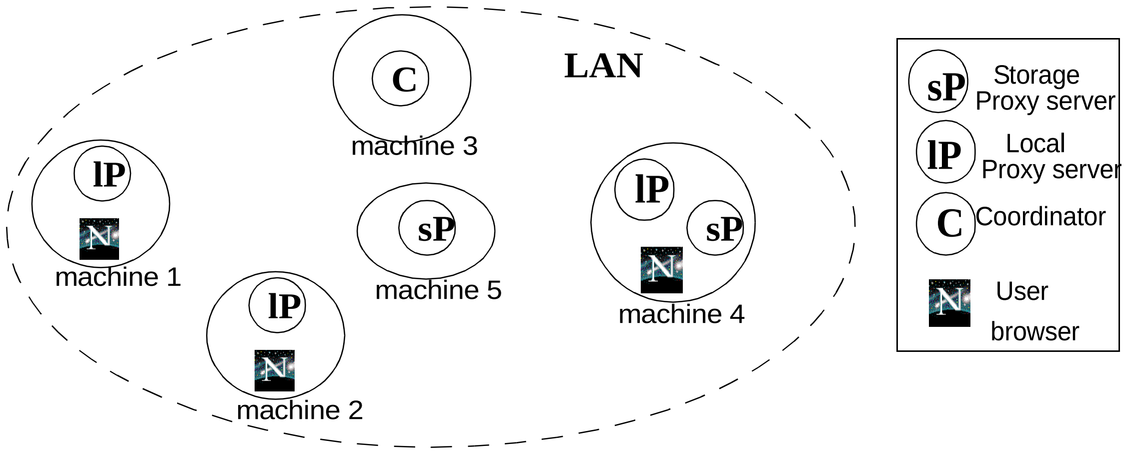


Figure 5.2: Proxy cluster in the modified architecture

In the revised architecture, I focus on load balancing between storage proxies since the load on the local proxies depends entirely on the activity of the clients they serve. I measured proxy load for using the following formula:

$$\text{histLoad}(P) = 0.5*(L_P/L) + 0.3*(R_{maxP}/R_{max}) + 0.2*(C_{maxP}/C_{max}) \quad \textbf{(EQ 3)}$$

where L_P is the number of bytes that have passed through proxy P in the past hour, L is the total number of bytes that have passed through the entire system, C_{maxP} is the peak number of connections on P in the past hour, C_{max} is the total peak number of connections on all the proxies, R_{maxP} is the peak bandwidth used by these connections on P, and R_{max} is the aggregate peak rate of all proxies in the system within the past hour. The histLoad metric calculates the load on a particular proxy *relative* to its counterparts, hence L_P, C_{maxP}, and R_{maxP} are normalized by the aggregate values L, C_{max}, and R_{max}, which are computed over all of the proxies. The weighting for the individual terms in the equation are empirical and give maximum emphasis to the recent number of bytes that passed through the proxy since ultimately, the goal is to minimize the variations in byte traffic of the individual proxies within a cluster. The instantaneous rates and connection terms are given less precedence in the load measure but are included, since it is also desirable to detect and prevent sudden fluctuations.

In order to test the modifications, I used a revised version of MiddleSim to evaluate LRU and *histLRUpick*, which uses the *HistLoad* metric. The former provides an example of the load imbalance typically seen when standard cache replacement algorithms are used, whereas the latter algorithm was shown by the previous investigations to be effective and was designed to balance the proxy load. I maintained the block size to be 1 Mbyte and constrained the global cache size to 44*50 Mbytes overall with 44 storage proxies serving 50 Mbytes each. I kept this global size since it achieved the highest byte hit rate of all the configurations tested and hence maximized inter-proxy activity. I also investigated the effect of reducing the number of proxies (but increasing the proxy size proportionately) on individual proxy load, evaluating the 22*100 Mbyte and 11*200 Mbyte configurations. I report my findings in the next section.

5.3.2 Results and Discussion

In order to view dynamic performance of MiddleMan, I plotted the temporal variations of certain system states for each cache configuration. The parameters displayed were:

- *Proxy Connection:* graphs the number of connections currently in the proxy system together with the *max connection*, the maximum number of connections currently at a proxy. Max connection is plotted on the *negative axis* for ease of comparison with the total number of connections in the system. Essentially, this plot indicates the current load imbalance. A well balanced proxy system will have a relatively small maximum connection, even if the total number of connections in the system is high. I define a well balanced system as one that evenly distributes load across proxies. At any given time, for example, the busy proxies in such a system will have similar loads.

- *Cache Rate*: A figure of this type compares the total rate of all the user connections served by the proxies and the WWW server against the bandwidth of the connections served by the proxies only. Essentially, a cache rate figure illustrates how well the cache performs with time. The system performs well if the cache bandwidth is close to the total bandwidth.

I utilize these metrics to investigate the dynamic cache performance, load behavior, and the effects of reducing the number of proxies in the MiddleMan architecture. I report on these analyses in the next three subsections.

5.3.2.1 Dynamic Cache Performance

Figures 5.3 and 5.4 graph the performance of the proxy cache with time for the *cdt* trace[2]. Total bandwidth of the files served by the system at any point in time is given by the black regions in the plot. The gray areas show the bandwidth served from the caches. A figure where the gray regions dominated over the black implies good caching performance. The brief all-black region at the very beginning of the trace in both figures 5.3 and 5.4 (day 1) indicate a “warm-up” period where the proxies are filling their caches with files. Afterwards, the cache accounts for a significant percentage of video traffic in both cases. MiddleMan tracks the requests closely even in worst case scenarios such as day 140 where twenty four simultaneous connections were requested, resulting in a peak system bandwidth of of about 1100 KBytes/sec and a corresponding cache rate of about 9300 KBytes/sec. The lack of cache activity during days 118-130 is

[2] I do not report cumulative byte hit rates as they remain the same as in the previous architecture.

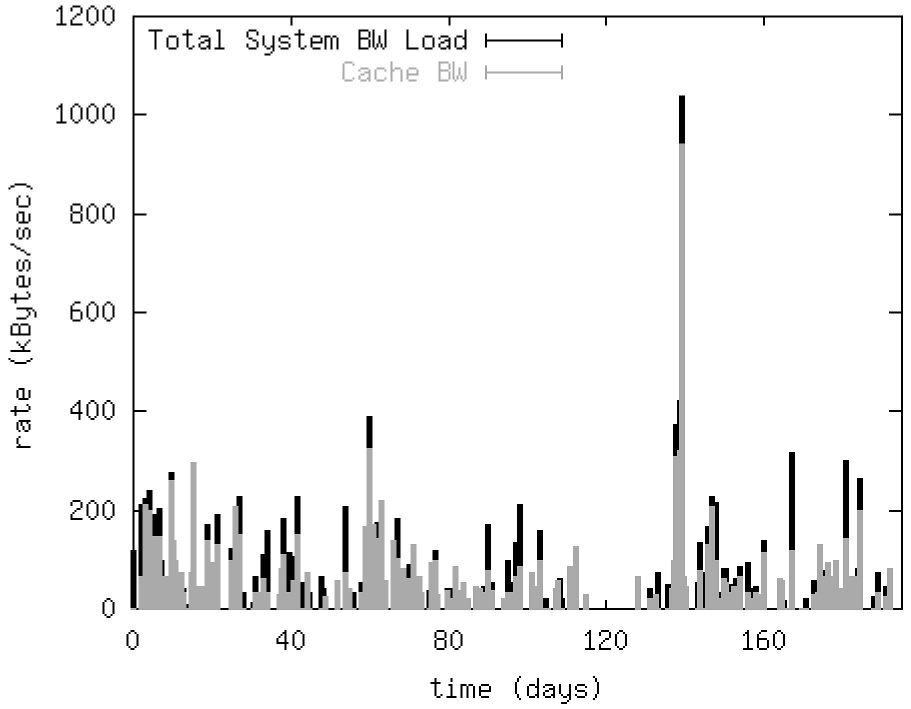


Figure 5.3: Cache Rate, LRU, 44*50 MByte

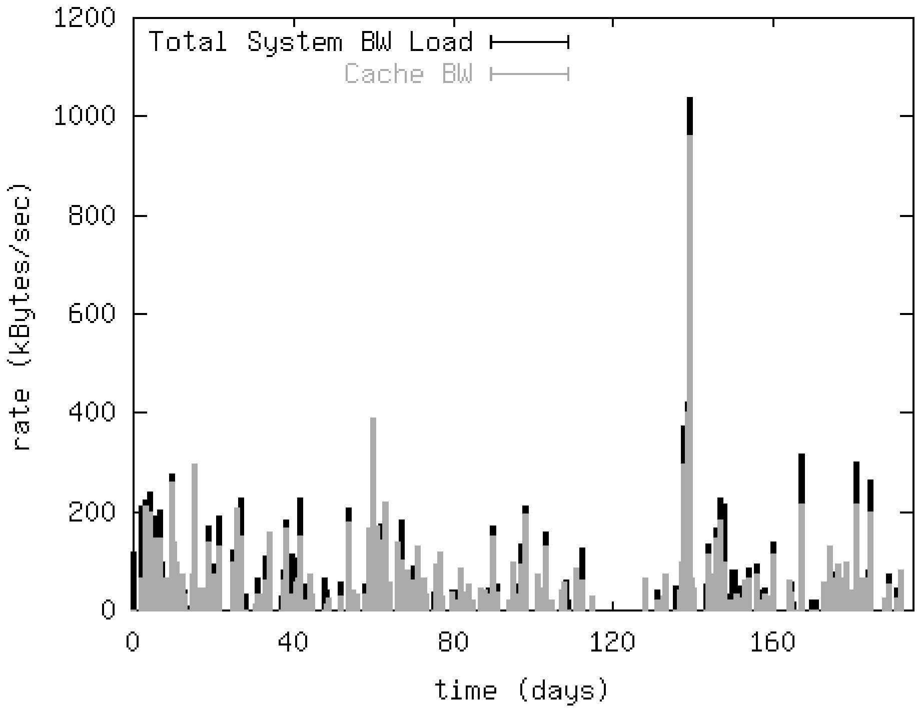


Figure 5.4: Cache Rate, histLRUpick, 44*50 MByte

due to Christmas break. Overall, the day to day differences between the caching performance of the two algorithms are minimal.

5.3.2.2 Load Behavior

Figure 5.5 summarizes the normalized standard deviation of the *cumulative* byte traffic of all the proxies under LRU and HistLRUpick. The figure shows that the latter algorithm drastically reduces the inter-proxy standard deviation. Figure 5.6 presents an alternate approach to displaying load imbalance. Using the *campus* trace, and configuration, it plots the most aggregate bytes served by a proxy and the least bytes served by a proxy in three configurations using LRU and histLRUpick. Under LRU, there is a great difference between the most and least loaded proxies. However, in each case, HistLRUpick reduces the overall

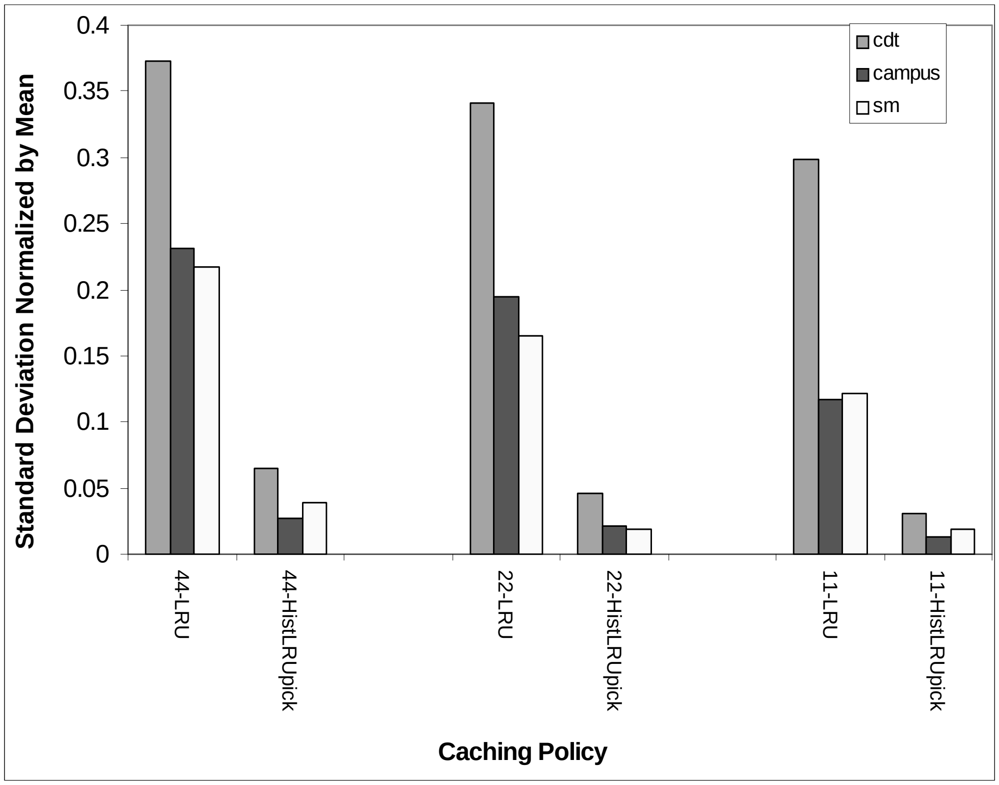


Figure 5.5: Normalized standard deviation for all the configs/traces

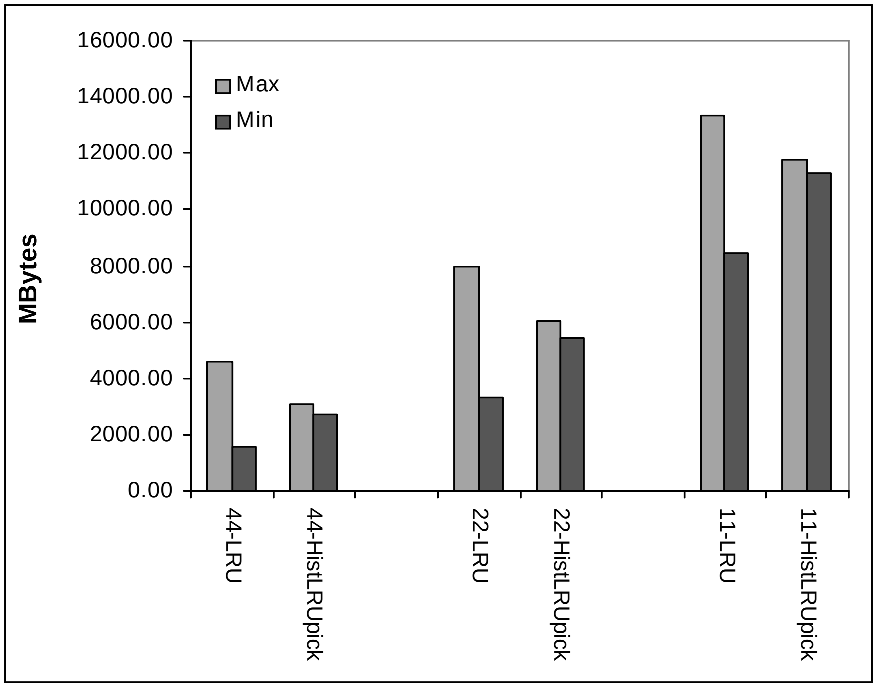


Figure 5.6: Overall max/min load report for *campus* trace

disparity by reducing the traffic on the heavily loaded proxy and increasing traffic to the lightly loaded one.

The dynamic load variations for the system with LRU and HistLRUpick is illustrated via their proxy connection plots (figures 5.7 and 5.8 respectively). The black region in the plots show the total number of connections served by the proxy at any given time whereas the gray areas display the maximum number of connections on a proxy in the system at the same instant, on the negative axis. Ideally, we would like the gray areas to be small compared to the black regions. This would imply that since the most loaded proxy was not heavily burdened, the remaining load was distributed over the other proxies. We would also prefer the maximum connection plot to be relatively smooth. Otherwise, excessive fluctuations would hint at sudden load changes on proxies, an undesirable property for a load balancing algorithm.

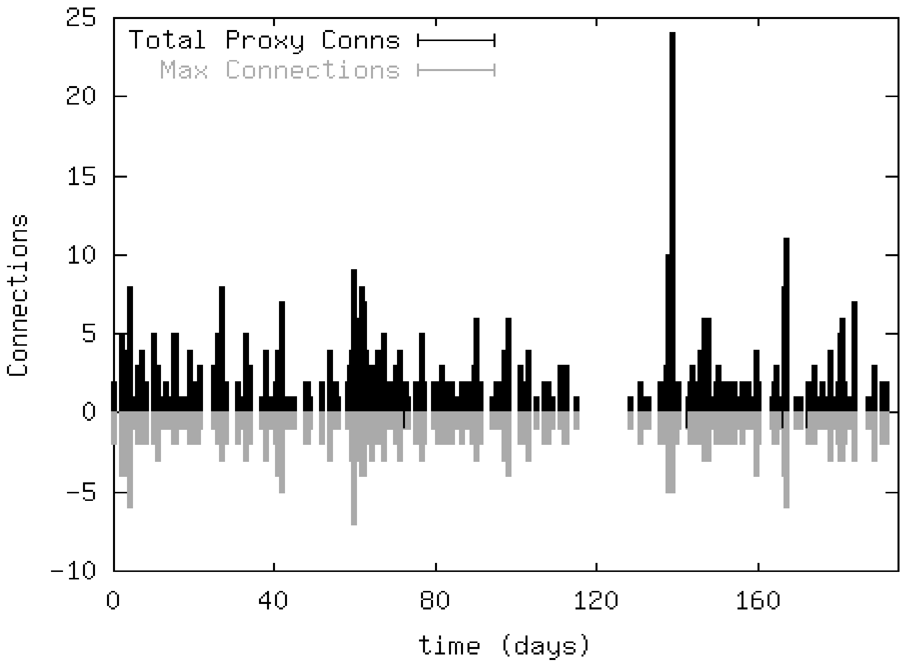


Figure 5.7: Proxy Connection, LRU, 44*50 MByte

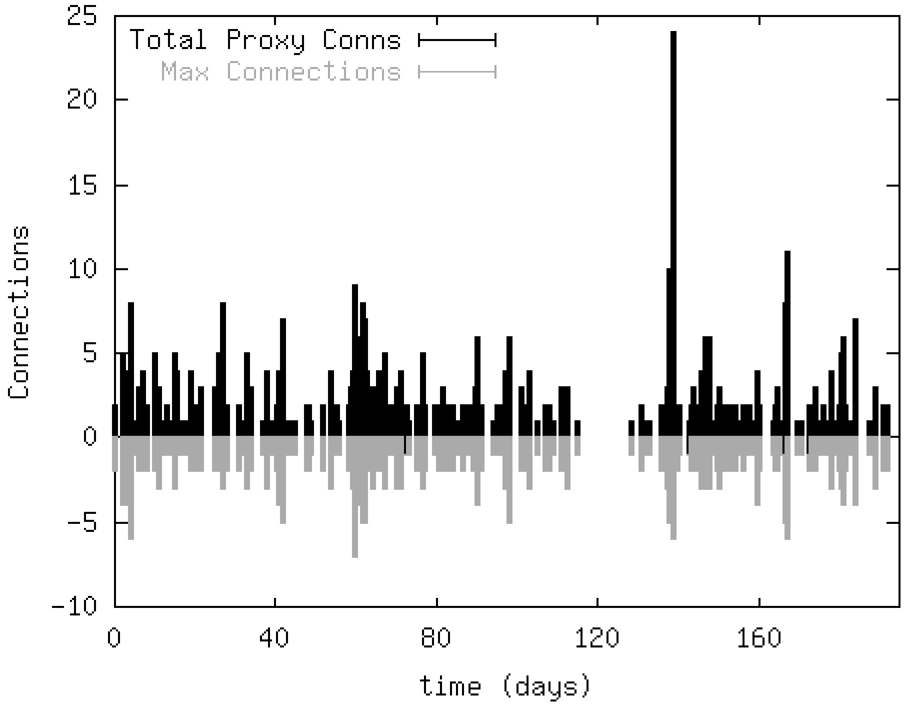


Figure 5.8: Proxy Connections, LRU, 22*100 MByte

Comparison of figures 5.7 and 5.8 shows that with the exception of day 140, HistLRUpick produces a smoother max connection plot that is more bounded than LRU. On day 140, both LRU and HistLRUpick have the same maximum number of connections (5) on their maximally loaded proxy. However, the peak max values of LRU exceed this threshold a number of times (on days 5, 40, 60, and 170, for instance) whereas HistLRUpick always stays below (or in the case of day 140, equal to) this cutoff value.

5.3.2.3 Decreasing the Number of Proxies

Figures 5.9 - 5.12 show the proxy connection plots for the 22*100 and 11*200 scenarios. I do not report on the caching performance as they remain similar due to the total global cache space still being the same. The figures for the 22*100 (5.9, 5.10) and 11*200 (5.10, 5.11) scenarios show trends similar to their 44*50 counterparts. However, the graphs also illustrate how proxy loads are affected when the number of storage proxies in the system are reduced -- the maximum proxy connection values increase in the system. With the exception of day 140, HistLRUpick is still able to smooth and bound the disparities better than LRU although its effectiveness drops with the number of storage proxies. How histLRU would behave under sustained heavy load remains an open question.

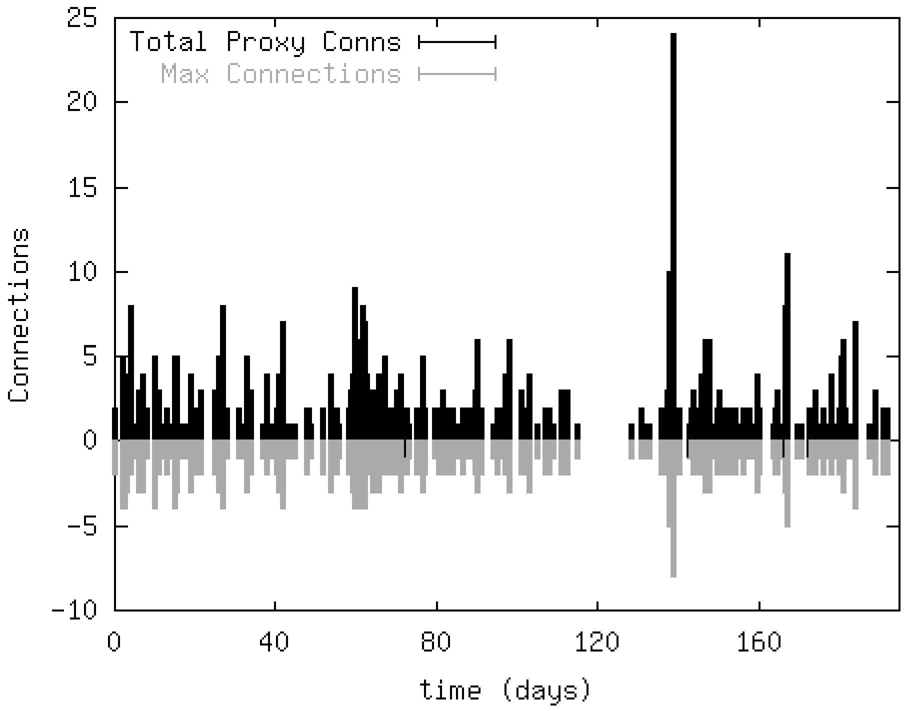


Figure 5.9: Proxy Connections, histLRUpick, 22*100 MByte

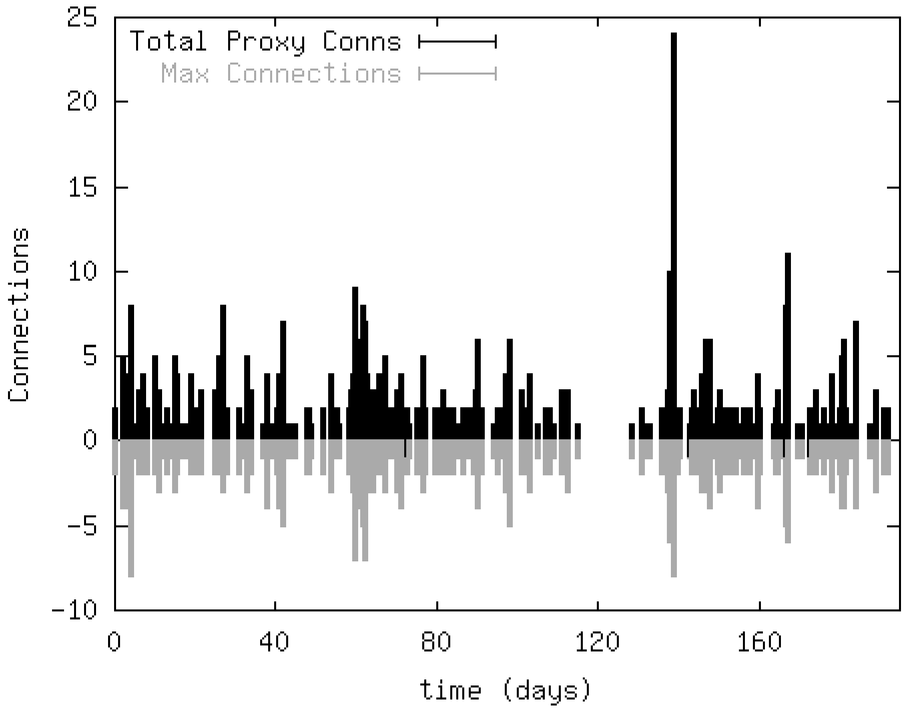


Figure 5.10: Proxy Connections, LRU, 11*200 MByte

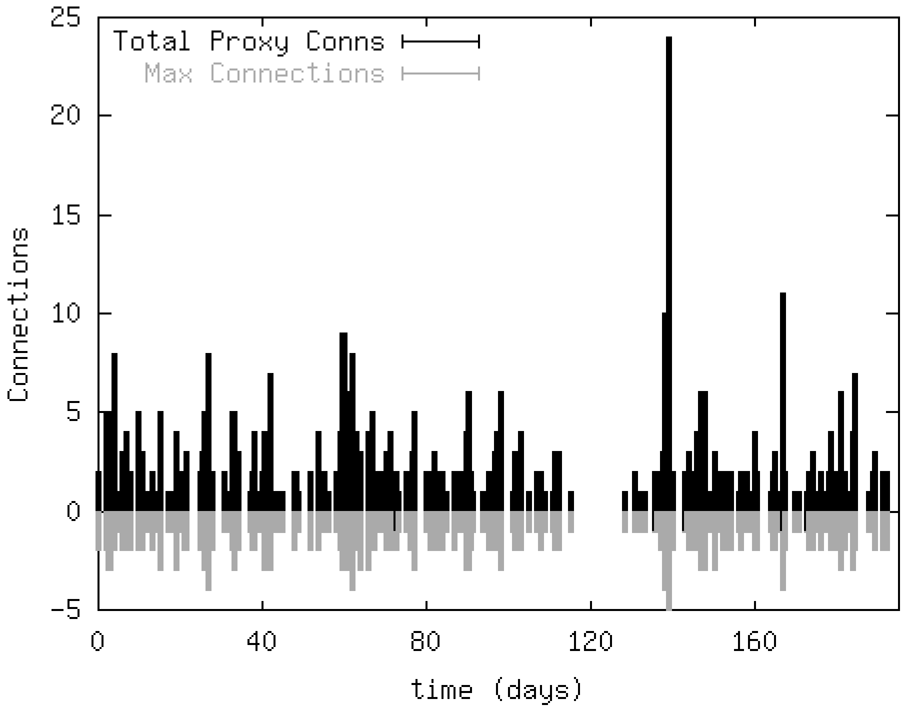


Figure 5.11: Proxy Connection, histLRUpick, 44*50 MByte

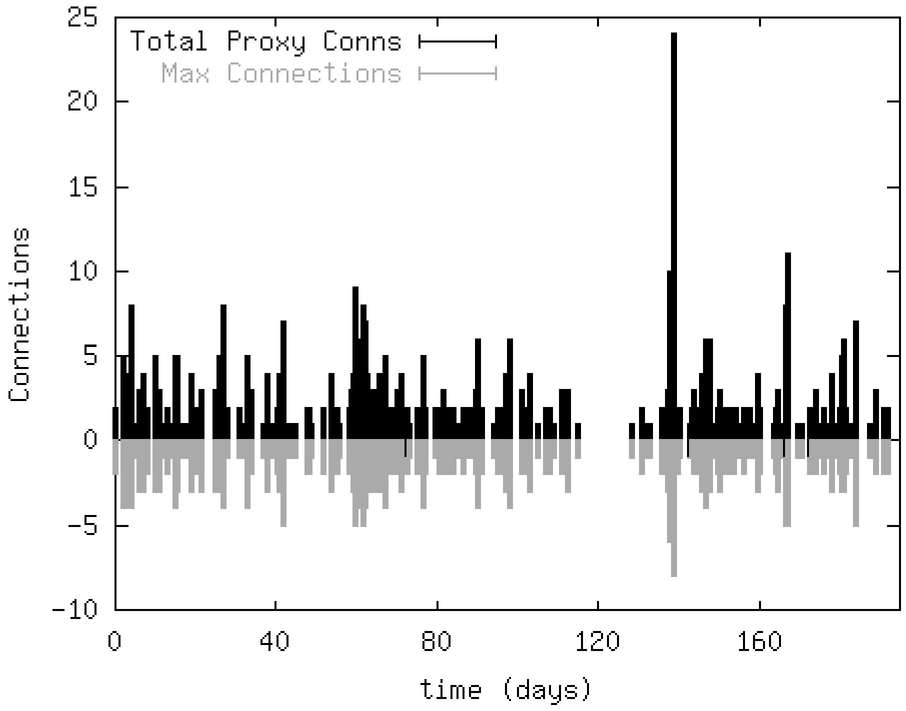


Figure 5.12: Proxy Connections, histLRUpick, 11*200 MByte

5.4 Block Size Variation Analysis

Table 5.2 illustrates the performance of the new MiddleMan architecture

Table 5.2: MiddleMan Performance With Increased Block Size

	cdt	***sm***	***campus***
1 Mbyte block size			
byte hit rate	65.32%	74.06%	76.26%
normalized std. dev.	0.3726	0.2284	0.2293
Max proxy bytes	3.234 Gbytes	4.151 GBytes	4.442 GBytes
Min proxy byes	0.098 Gbytes	1.226 GBytes	1.556 GBytes
5 Mbyte block size			
byte hit rate	64.95%	73.08%	76.12%
normalized std. dev.	0.3968	0.266	0.2382
Max proxy bytes	3.285 GB	4.14 GB	4.529 GB
Min proxy byes	0.098 GB	1.006 GB	1.471 GB
10 Mbyte block size			
byte hit rate	64.42%	73.27%	75.82%
normalized std. dev.	0.43	0.3165	0.2322
Max proxy bytes	3.453 GB	4.465 GB	5.071 GB
Min proxy byes	0.098 GB	0.426 GB	1.764 GB

with a proxy configuration of 44*50, LRU cache replacement policy and block sizes of 1 MByte, 5 MBytes, and 10 MBytes respectively. Increased block sizes lead to smaller byte hit ratios and increased load imbalance, as indicated by the progressively larger standard deviations and disparities between the maximum and minimum loaded proxies. The drop in byte hit rates is due to cache space being utilized less effectively. Similarly, larger block sizes provide less opportunities for redistributing cache load, thus resulting in more skewed systems. However, the overall drop in performance is small indicating that in situations with high client activity, it might be worthwhile to increase block sizes to reduce the message processing load on the coordinator and better buffer against possible inter block switching delays.

Another possibility not explored in the current investigation is to allow a number of block sizes to simultaneously co-exist depending on the needs of the user and streaming requirements of the video file. For instance, bandwidth intensive video files might be more efficiently served with large block sizes since this amortizes the proxy-coordinator message processing overhead costs.

5.5 Conclusion

In this chapter, I have investigated the performance of MiddleMan. I found LRU-k to provide the highest hit rates but a variation, HistLRUpick, in conjunction with an alteration to the system architecture yielded good hit rates as well as effective load balancing. A relatively small global cache size of 2.14 GBbytes (44*50 Mbytes, about 13.7% of total file sizes) resulted in very high byte hit rates. I also found that a larger number of proxies were preferable to a smaller number of proxies even though overall global cache size would remain the same. More proxies resulted in smaller peak loads in each proxy and hence, better load balancing.

Chapter 6

Transcoding from MPEG to JPEG

6.1 Introduction

The results from chapter 2 show MPEG to be probably the most popular single video format, even though QuickTime files outnumber MPEG. QuickTime is a collection of codec technologies none of whom individually exceed MPEG in acceptance. Given the popularity of MPEG and the rising acceptance of related compression schemes such as H.261, H.263 and MPEG-4, in this chapter I concentrate on fast video transcoding of MPEG to M-JPEG. Integration of such a capability into a caching proxy is useful as it allows for more leeway in client load control. A proxy or a client can drop arbitrary frames if necessary in response to sudden client loads or local network bottlenecks. This is not possible with the other formats.

Transcoding is also potentially useful in other applications:

- Video editing: random access of video data is important and M-JPEG has this property, unlike MPEG which is often preferred for storage.
- Video processing: video is usually decompressed before it can be processed. However, *compressed domain processing* techniques can operate on video without fully decompressing it. Many of these techniques operate on M-JPEG data [55, 54, 15].

In this chapter, I develop a compressed domain transcoder (*CDTC*) for converting MPEG-1 to M-JPEG that is 1.5 to 3 times faster than its spatial domain counterpart. Additional speedup is possible at the expense of image quality. The techniques are optimized for MPEG-1 but can be adapted for MPEG-2 and H.261/H.263. I concentrate on the CDTC since much work has already been done elsewhere on compressed domain processing.

This chapter assumes the reader is familiar with MPEG-1 and JPEG compression [25, 27, 60]. Nonetheless, in order to introduce basic terminology and concepts, I briefly review these algorithms in section 6.2. To identify the problems associated with transcoding, I describe spatial domain transcoding in section 6.3. I provide more details on my CDTC in section 6.4 and also show how I optimized processing while maintaining good picture quality. In sections 6.5 and 6.6 I compare the performance of the spatial and compressed domain methods. I outline related work in section 6.7, and finally, in section 6.8, I present some conclusions and directions for future work.

6.2 A Brief Review of JPEG and MPEG-1

To understand the compressed domain transcoder, a working knowledge of JPEG[1] and MPEG is necessary. This section briefly reviews JPEG and MPEG. My purpose in this exposition is to remind the reader of the steps in the algorithms and to name these steps for purposes of discussion. A more detailed discussion of these standards are available elsewhere [25, 27, 60]. To simplify the discussion, we only consider gray-scale video.

[1] Motion-JPEG applies the JPEG algorithm to each frame in a video sequence.

6.2.1 JPEG

The sequential (baseline) JPEG algorithm, shown in figure 6.1, consists of the following steps:

1. Decomposition: The original image is divided into 8x8 *blocks*. These blocks are processed in row-major order by the following steps.
2. DCT: Each block is transformed using the Discrete Cosine Transform (DCT). The transformed block has the signal energy concentrated into a few lower order coefficients. The (0,0) element of the transformed block is called the *DC component*, and the other 63 elements are *AC components*.
3. Scaling: Each element in the transformed block is divided by the corresponding elements in an 8x8 *quantization table* (QT).
4. Rounding: The scaled coefficients are rounded to the nearest integer. Steps 3 and 4 together are called *quantization*.
5. Zig-zag mapping: Each quantized 8 x 8 block is converted to a 64 element *vector* using a fixed one to one mapping.
6. Run Length Encoding: Strings of zeros in the AC elements of the quantized vector are run length encoded. The block, at this stage, is called a *semi-compressed (SC)* block. An SC-block consists of a DC element plus several *(run length, AC value)* pairs.
7. DPCM: The DC coefficient of each SC block is encoded as *difference* from the DC coefficient of the previous block in the sequence.
8. Huffman Coding: The SC block is converted to a bitstream via Huffman encoding.

An important property, for our purposes, is that some of these steps can be transposed. For example, we can swap steps 4 (rounding) and 5 (zig-zag mapping). With this exchange, the first three steps, DCT, scaling and zig-zag mapping, are all linear. Thus, they can be combined into one linear operation, a fact I utilize in section 6.4. Also, note that SC-blocks are simply sparse representations of the scaled DCT block. The compressed domain algorithms will operate on SC-blocks, and we represent a JPEG image as an array of SC blocks.

Decompression is the reverse of compression. The bitstream is *entropy decoded* and the DC value of the previous block is added to the recovered difference value to produce an SC block. The SC block is de-*zigzagged* into an 8 x 8 quantized block. This block is multiplied by the QT (the *multiplication* step) and the inverse DCT transform (IDCT) is applied.

6.2.2 MPEG

The MPEG video standard is designed to compress sequences of images (also called *frames*). In MPEG-1 video (hereafter called MPEG), each frame is divided into 16 x 16 pixel *macroblocks*. Each macroblock contains six 8 x 8 pixel *blocks*, four from the luminance channel and one from each chrominance channel. As with JPEG, we ignore the chrominance channel for simplicity of discussion. Each macroblock can be compressed in several ways. Depending on the method used, the macroblock is called an I, P, B, or Bi macroblock[2]. MPEG also defines three types of frames, I, P and B, which we discuss in turn and summarize in table 6.1.

[2] Two other types of macroblocks, called D and skipped, are also defined by MPEG. We ignore them in this paper for simplicity.

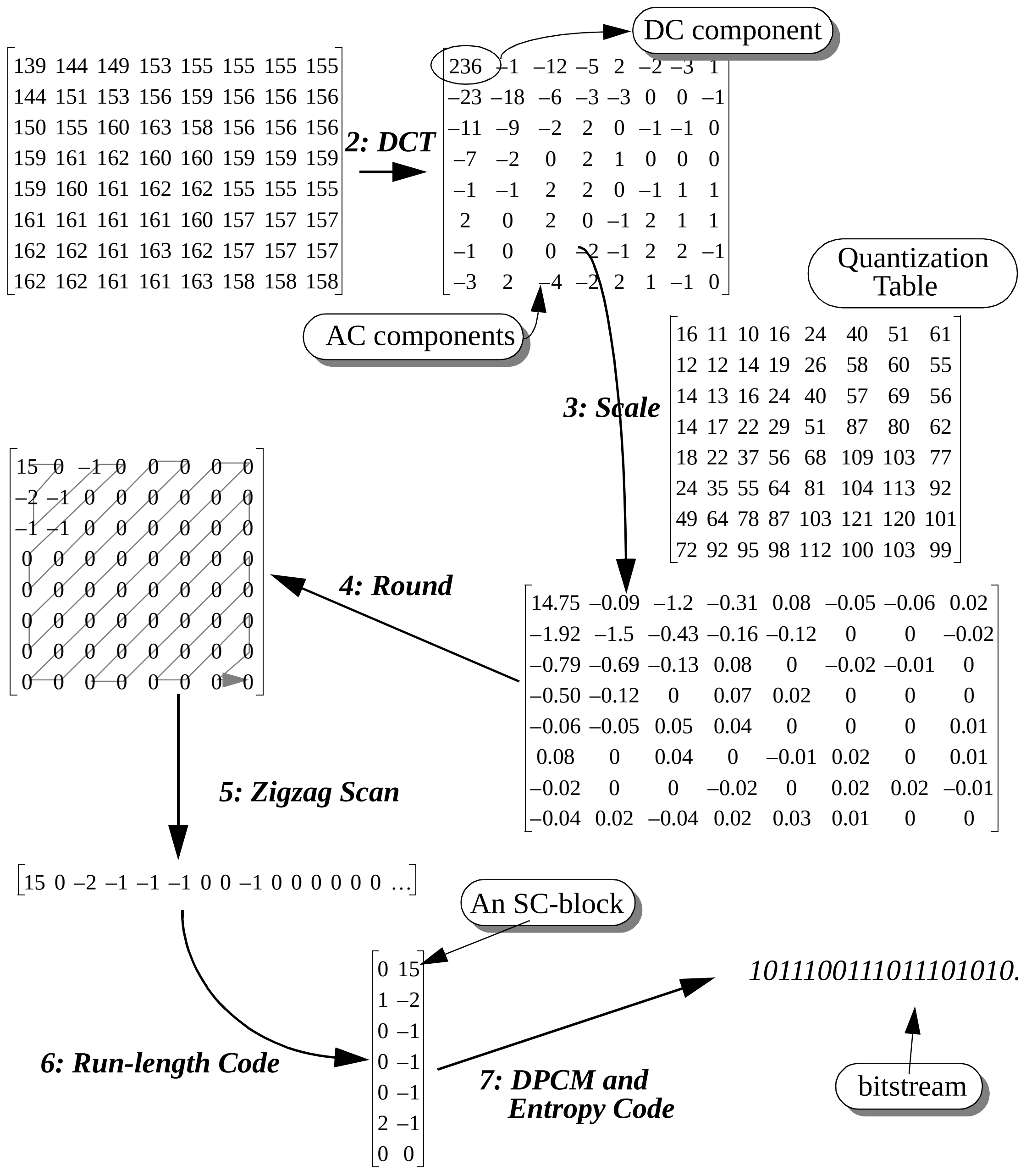


Figure 6.1: Block Encoding

I frames contain only I macroblocks. The four blocks in an I macroblock are compressed using an algorithm nearly identical[3] to JPEG. In other words,

[3] The bitstream syntax, entropy encoding technique, and quantization tables are different, as is the encoding order of the image blocks.

the blocks are converted to SC-blocks and then compressed using Huffman coding.

P frames exploit the temporal redundancy of video whereby adjacent frames are likely to be similar. A P frame relies on a *reference frame* prior in the sequence called a *past* reference frame. The past reference frame is the most recent I or P frame in the sequence. A P frame consists of either I or P macroblocks. A P macroblock is a *motion vector* and a *residual* macroblock. During decompression, the motion vector is used to extract a *predicted* macroblock from the reference frame, and the residual macroblock is added to the predicted macroblock to produce the macroblock in the output image. The residual macroblock is compressed like the I macroblock, and the motion vectors are also encoded in the bitstream using Huffman coding.

A B frame is similar to a P frame, but uses two reference frames, one from later in the sequence (a *future* reference frame) and one from earlier on in the sequence (a *past* reference frame). As with P frames, these reference frames are the nearest I or P frames in the sequence (see figure 6.2). A B frame can contain I, P, B, or Bi macroblocks. B macroblocks are identical to P, except they use the future reference frame rather than the past reference frame. Since B macroblocks are so similar to P macroblocks, the techniques we use to deal with the two are identical. Hence, we refrain from discussing B macroblocks for the remainder of this chapter.

A Bi macroblock consists of two motion vectors and a residual macroblock. These motion vectors are used to extract the two predicted macroblocks (one from each reference frame), which are averaged and added to the residual macroblock to produce the result. The residual macroblock is compressed to an SC-block and Huffman coded along with the two motion vectors.

Table 6.1: Summary of MPEG frames and Macroblocks

Frame Type	Frame Composition		Macroblock Type	Macroblock composition
I frame	I type macroblock		I	4 SC blocks
P frame	I, P		P	4 SC + motion vector (*mv*)
B frame	I, P, B, Bi		B	4 SC + mv
			Bi	4 SC + 2 mv

Table 6.1 summarizes the composition of MPEG frames and macroblocks. When the blocks in a macroblock are semi-compressed, we call the macroblock an *SC macroblock*. Similar to JPEG, a frame can be represented as a two dimensional array of macroblocks. If these macroblocks are semi-compressed, the frame is called a *SC frame*. The use of reference frames in I, P, and B frames leads to the interframe dependencies shown in figure 6.2. It is these dependencies that complicate random access.

6.3 Spatial Domain Transcoding

Having sketched MPEG and JPEG, I now turn to the problem of converting MPEG to Motion-JPEG (MJPEG). The most straightforward approach is to

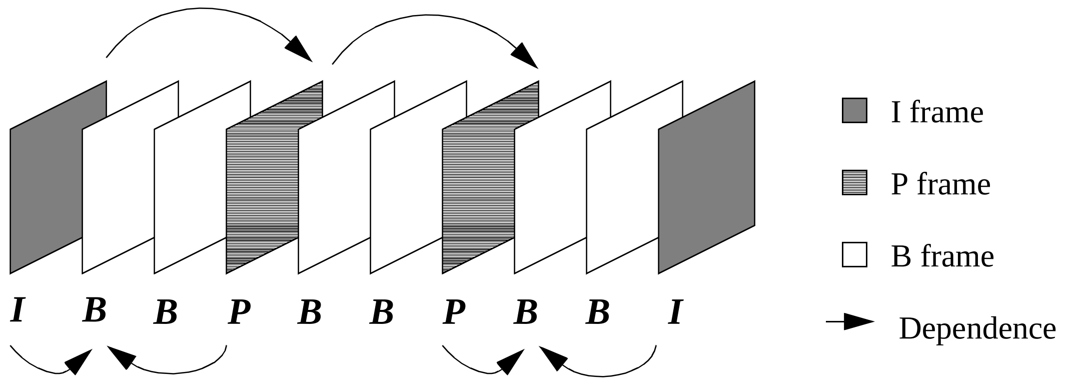


Figure 6.2: Inter-frame dependencies in MPEG

decompress the source bitstream completely and compress the result using JPEG. I call this technique method I. It provides a benchmark against which other techniques can be compared.

Method I can be optimized in several ways. The first optimization is to note that I macroblocks can be converted to JPEG blocks simply by requantizing the SC-blocks in the I-frame. Or, better yet, we can redefine the quantization tables in the JPEG bitstream to be the same as used by MPEG and just entropy encode the SC-blocks in the I macroblock directly (if we slightly reorder the blocks). An important detail to remember is that MPEG uses variable quantization and standard JPEG does not, so the coefficients in the SC block may need to be rescaled, but the cost of this rescaling is minimal.

Transcoding P and Bi macroblocks to JPEG blocks is more complex because a predicted macroblock(s) must be extracted from the reference frame(s). The problem is illustrated in Figure 6.3. As we can see from the figure, up to four reference blocks may be needed to reconstruct a predicted block. These reference blocks, which may be contained in different macroblocks, must be decompressed to construct the predicted block. The residual block must also be decompressed before it is added to the predicted block, and the result compressed.

Extracting the predicted blocks is expensive, since it requires decompressing the relevant macroblocks in the reference frame, but a few optimizations can be made. Experiments reveal that some macroblocks in a reference frame are decompressed multiple times, since they can be referenced in multiple B frames. We therefore decompress the macroblocks in the reference

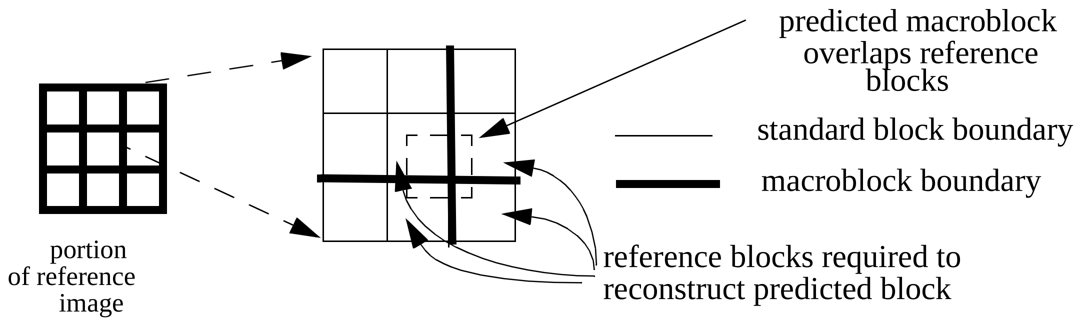

predicted macroblock overlaps reference blocks
standard block boundary
macroblock boundary
reference blocks required to reconstruct predicted block
portion of reference image

Figure 6.3: Source Block Extraction Problem

frames as needed and cache the decompressed macroblocks in case we need them later.

Another optimization is to add the predicted and residual macroblocks in the compressed domain [15]. In normal processing, the residual macroblock is decompressed, added to the predicted macroblock, and the result is compressed. But, if the predicted macroblock is first converted to an SC macroblock, it is possible add the residual SC macroblock directly, avoiding the decompression step.

The modified transcoding method, with these three optimizations (I-frame conversion, caching, and adding the residual in the compressed domain), is called Method II. Method II represents a reasonable effort to improve the performance of a spatial domain transcoder with relatively simple compressed domain techniques. Experiments show that the DCTs and IDCTs required by P/B macroblock extraction are the bottleneck in transcoding speed for Method II. My next approach, therefore, was to perform predicted block extraction in the compressed domain. The next section describes this approach in detail.

6.4 Method III - Compressed Domain Transcoding

The bottleneck in Method II is extracting predicted macroblocks from reference images. This section shows how to perform this operation in the compressed domain. I first show how to decompose source block extraction into steps that can be implemented on the SC-blocks.

As evident in figure 6.3, we can extract a predicted macroblock if we can align the relevant macroblocks in the reference frame to a macroblock boundary. Another way to think of this macroblock alignment is that it involves *translating* the individual blocks and combining them, as shown in figure 6.4. In this figure, the block f_{00} is translated by (dx, dy). We use the notation $\boldsymbol{T_{dx,dy}f_{00}}$ to represent this translated block. The black region in $\boldsymbol{T_{dx,dy}f_{00}}$ are pixels not contained in f_{00}, so we set them to zero. If the other three blocks are similarly translated, and the results added, one block in the predicted macroblock is extracted.

I now show how to perform these operations (translation and addition) directly on the SC-blocks. The SC blocks can be added directly, we can perform this entire operation on SC blocks if we can perform translation on SC blocks.

6.4.1 Deriving The Compressed Domain Translation Operator

Previous work [54, 55] showed how to perform a wide variety of operations, including translation, directly on SC blocks. Briefly, let F_{00} be the SC-block corresponding to f_{00}. Formally, $\boldsymbol{T_{dx,dy}}$ is then a matrix, F_{00} a sparse vector, and translation of F_{00} is accomplished by matrix multiplication. The result of this mul-

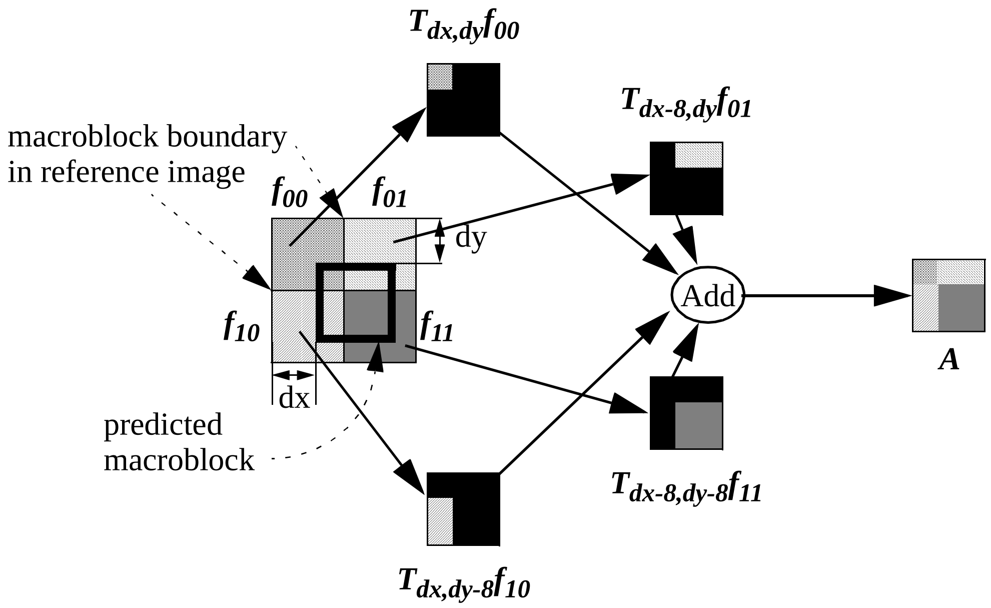


Figure 6.4: Block alignment using translation

tiplication is a 64 element vector that represents the translated block after the DCT, scaling, and zig-zag scan steps (steps 2, 3, and 5) of the JPEG algorithm have been performed. This vector is converted to an SC block using steps 4, 6, and 7 (rounding, DPCM, and run-length encoding) of the JPEG algorithm.

The DCT transform, quantization and zig zag scan steps in the block encoding process can all be modelled as matrix operations. We define the forward DCT (FDCT) as a function *D* which operates on an 8x8 pixel block *f*. *D* is a 8x8x8x8 matrix.

$$FDCT(\alpha, \beta) = \sum_i \sum_j D(i, j, \alpha, \beta) f(j, i) \qquad \textbf{(EQ 4)}$$

where

$$D(i, j, \alpha, \beta) = \left(\frac{1}{4}\right) C(i, \alpha) C(j, \beta) \qquad \textbf{(EQ 5)}$$

$$C(i, \alpha) = A(\alpha)\cos\frac{((2i+1)\alpha\pi)}{16} \qquad \text{(EQ 6)}$$

$$A(\alpha) = \begin{cases} \frac{1}{\sqrt{2}} & \alpha = 0 \\ 1 & otherwise \end{cases} \qquad \text{(EQ 7)}$$

Similarly, the zig-zag operator *Z* maps an 8x8 block to a 64 element vector:

$$Z(\gamma, \alpha, \beta) = \begin{cases} 1 & zigzag[\alpha, \beta] = \gamma \\ 0 & otherwise \end{cases} \qquad \text{(EQ 8)}$$

The quantization operator *Q* divides each element *i,j* in an 8x8 array by q(i,j).

$$Q(k, \gamma) = \delta(\gamma, k)/q(k) \qquad \text{(EQ 9)}$$

The whole block encoding process is given by:

$$F(k) = \sum_{\gamma}\sum_{\alpha}\sum_{\beta}\sum_{i}\sum_{j} Q(k, \gamma)Z(\gamma, \alpha, \beta)D(\alpha, \beta, i, j)f(i, j) \qquad \text{(EQ 10)}$$

From this we can define the JPEG operator, *J* to be:

$$J(k, i, j) = \sum_{\gamma}\sum_{\alpha}\sum_{\beta} Q(k, \gamma)Z(\gamma, \alpha, \beta)D(\alpha, \beta, i, j) \qquad \text{(EQ 11)}$$

J is a 64x8x8 matrix. By a similar derivation process, we arrive at $\bar{J}$, the reverse JPEG operator.

$$\bar{J}(u, v, l) = \sum_{\alpha}\sum_{\beta}\sum_{\gamma} \bar{D}(u, v, \alpha, \beta)\bar{Z}(\alpha, \beta, \gamma)\bar{Q}(\gamma, l) \qquad \text{(EQ 12)}$$

An image translation is a linear mapping operation. To translate a block of pixels, *f*, by an offset of *dx* in the direction of increasing x and *dy* in the direction of increasing y, we would say:

$$\hat{f}(x, y) = \begin{cases} f(x-dx, y-dy) & 8 > x \geq dx \quad 8 > y \geq dy \\ 0 & otherwise \end{cases} \quad \text{(EQ 13)}$$

The challenge is to perform the translation operation in the compressed domain. We know:

$$\hat{F}(l) = \sum_x \sum_y J(x, y, l)\hat{f}(x, y) \quad \text{(EQ 14)}$$

Applying equation 11 yields:

$$\hat{F}(l) = \sum_x^8 \sum_y^8 J(l, x, y) f((x-dx), (y-dy)) \quad \text{(EQ 15)}$$

Shifting indices, we get:

$$\hat{F}(l) = \sum_x^8 \sum_y^8 J(l, (x+dy), (y+dx)) f(x, y) \quad \text{(EQ 16)}$$

We substitute *f* by ($\bar{J}F$):

$$\hat{F}(l) = \sum_x \sum_y \sum_k J(l, (x+dy), (y+dx)) \bar{J}(x, y, k) F(k) \quad \text{(EQ 17)}$$

J, $\bar{J}$, *dx* and *dy* are known quantities. Hence we can precompute the translation matrix *T*:

$$T_{dx,\,dy}(l, k, dx) = \sum_{x}\sum_{y} J(l, (x + dy), (y + dx))\bar{J}(x, y, k) \qquad \textbf{(EQ 18)}$$

$T_{dx,\,dy}$ is the translation frequency domain transform matrix for a given *dx* and *dy*.

Figure 6.5 shows a slightly different decomposition, where we align macroblocks by translating on each axis separately. The intermediate blocks $\mathbf{G_0}$ and $\mathbf{G_1}$ and the predicted block **A** are given by the equations

$\mathbf{G_0} = \mathbf{T_{dx}F_{00}} + \mathbf{T_{dx\text{-}8}F_{01}}$

$\mathbf{G_1} = \mathbf{T_{dx}F_{10}} + \mathbf{T_{dx\text{-}8}F_{11}}$

$\mathbf{A} \;\; = \mathbf{T_{dy}G_0} + \mathbf{T_{dy\text{-}8}G_1}$

Analysis shows that computing this decomposition is more efficient than computing the combined operation (figure 6.4) because the matrices associated with the translation along a single axis, $\mathbf{T_{dx}}$ and $\mathbf{T_{dy}}$, are quite sparse. We therefore focus on this case. In the discussion that follows, we describe the case of translating a block **F** by a positive amount *dx* along the *x* axis. Translating along the *y* axis and translating by negative values are similar.

Our goal is to compute the elements of a vector $\mathbf{G = T_{dx}F}$:

$$G[i] = \sum_{j=0}^{63} T_{dx}[i, j]F[j] \qquad \textbf{(EQ 19)}$$

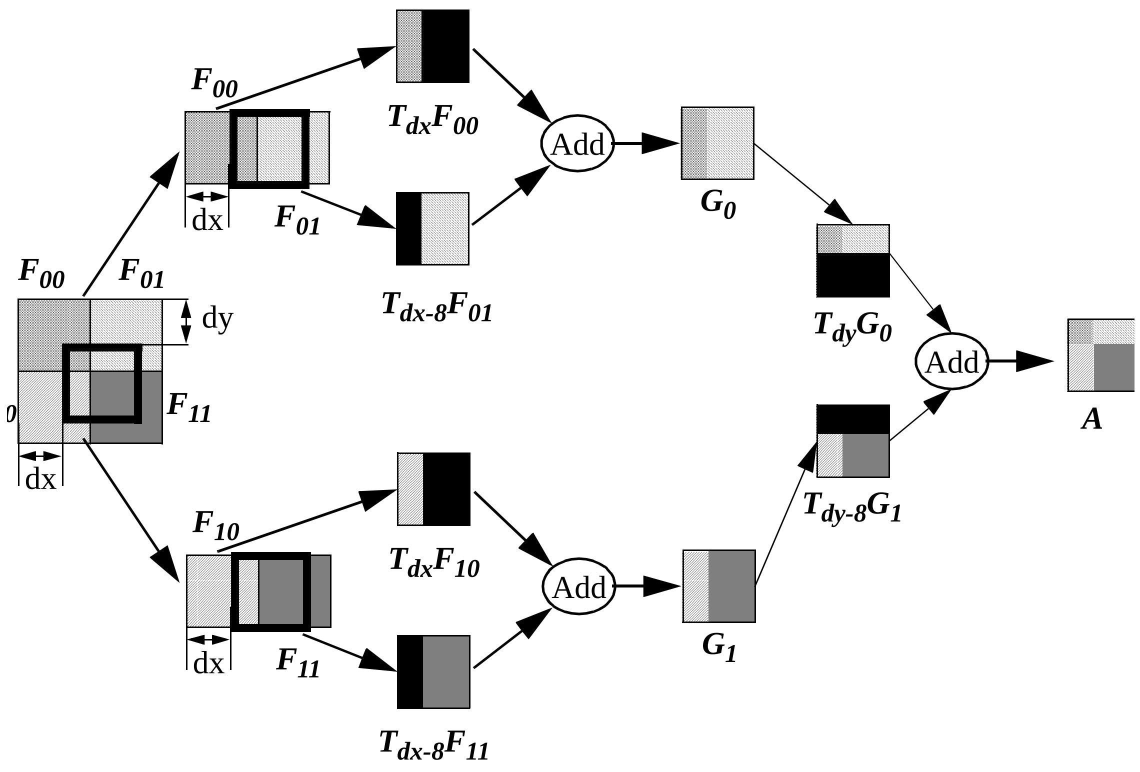


Figure 6.5: Translating on each axis separately

Following the development in [55], the elements of the matrix $T_{dx}[i,j]$ are given by:

$$T_{dx}[i, j] = \sum_{x=0}^{63}\sum_{y=0}^{63} J(i, x, (y + dx))\bar{J}(x, y, j) \qquad \textbf{(EQ 20)}$$

where J is a 3 dimensional matrix that converts an SC block from the source block and $\bar{J}$ performs the reverse computation. Note that scaling, zig-zag encoding, DCT and IDCT are folded into J and $\bar{J}$ and hence into $\mathbf{T_{dx}}$. We present code to compute $\mathbf{T_{dx}[i,j]}$ in the appendix. Since the parameter dx can only assume sixteen discrete values (dx = 0..7 plus half pixels), our task is to efficiently compute equation 19 for these sixteen values. We describe an efficient method in the next section.

6.4.2 Performance Optimization I - Dynamic Thresholding

The vector **G** in equation 19 is the product of a matrix $\mathbf{T_{dx}}$ and the vector corresponding to the SC block **F**. In the calculation of **G**, we can tolerate some error in the result. We can afford some error because the elements of **G** will be rounded off to the nearest integer (quantized) after the computation, as part of converting **G** to an SC block. Thus, introducing errors of, say, +/- 0.5 should not unduly affect the quality of the output. Suppose we fix an allowable error *maxerr* for the computation of each element of **G**, and that **F** has *n* non-zero elements. Computing an element *G[i]* requires *n* multiplies (and *n-1* adds) of the form $T_{dx}[i,j]*F[j]$. We call a product term *insignificant* if:

$$|T_{dx}[i,j]F[j]| < \frac{maxerr}{n} \qquad \textbf{(EQ 21)}$$

or

$$|T_{dx}[i,j]| < \left|\frac{maxerr}{nF[j]}\right| \qquad \textbf{(EQ 22)}$$

Intuitively, insignificant terms are terms in equation 19 that we can throw away without introducing too much error (i.e., more than *maxerr*). The central idea of our optimization is that we compute the product $T_{dx}[i,j]*F[j]$ only if equation 22 holds. Conceptually, we must check this condition before every multiply. However, since T_{dx} is a constant, it turns out there is an elegant way of doing this efficiently.

Consider the calculation of **G** in equation 19. Using the fact that **F** is typically sparse, we can calculate **G** using the following code fragment:

```
zero G[]
for each non-zero F[i] do
      for all j do
            G[j] += T[i,j]*F[i];
      endfor
endfor
```

If we unroll the inner loop using a switch statement, our code fragment looks like this:

```
zero G[]
for each non-zero F[i] do
      v = F[i];
      case i in
       0:
            G[0] += T[0,0]*v;
            G[1] += T[0,1]*v;
            G[2] += T[0,2]*v;
            ....
       1:
            ....
```

For a fixed translation *dx*, **T** is constant, so we can substitute T[0,0], T[0,1] by precomputed values. For example, suppose T[0,0]=0.3214, T[0,1]=-0.0027, T[0,2]=-0.271, T[0,3]=-0.027, and the remaining T[0,j] are zero. Our code fragment becomes:

```
zero G[] vector
for each non-zero F[i] do
      v = F[i];
      case i in
       0:
            G[0] += 0.3214*v;
            G[1] += -0.0027*v;
            G[2] += -0.271*v;
            G[3] += 0.1045*v;
            ....
```

Finally, we sort each case statement by the absolute value of the constant, smallest value at the top:

```
zero G[] vector
for each non-zero F[i] do
      v = F[i];
      case i in
       0:
            G[1] += -0.0027*v;
            G[3] += 0.1045*v;
            G[2] += -0.271*v;
            G[0] += 0.3214*v;
            ....
```

For a given value of *F[i]*, only some of the terms *T[i,j]*F[i]* are significant. For example, if *F[i]=1*, *maxerr=0.5*, and *n=3*, then by equation 22, any term with a coefficient *T[i,j]* less than *0.5/3*1 = 0.167* is insignificant (i.e., the first two terms in case 0 of the example above). Since the values are sorted, we know that all the preceding terms are insignificant as well. Thus, if we can jump to the first significant coefficient in the case statement, we will only evaluate the product terms required. We call this technique *dynamic thresholding*.

It is possible to implement dynamic thresholding in this case by computing an integral threshold, *intThresh*, which is a fixed point representation of the coefficient threshold (equation 22):

$$intThresh = \left\| \left\lceil \frac{maxerr}{nF[i]} \times 16 \right\rceil \right\| \quad \textbf{(EQ 23)}$$

We use *intThresh* as an index into a jump table that skips insignificant coefficients *T[i,j]*. Implementing this idea results in the following code:

```
zero G[]
n = number of non-zero elements in F[]
for each non-zero F[i] do
      v = F[i];
      intThresh = abs(ceiling(16*maxerr/(F[i]*n)));
      case i in
       0:
            case intThresh in
             0, 1:
                  G[1] += -0.0027*v;
             2:
                  G[3] += 0.1045*v;
             3:
                  G[2] += -0.271*v;
             default:
                  out[0] += 0.3214*v;
            ...
```

Note that this code can execute very efficiently. Once the initial jump is performed, the minimal number of product terms are computed as a sequence of multiply/add instructions. Experiments show that 20%-30% of the multiplies are avoided for typical values of *maxerr*. Since there is no interdependency of terms, these instructions can be pipelined resulting in a throughput of one cycle per term.

Since writing code of this nature is tedious, I developed a code-generator to write the required functions. My generator produces a procedure for each offset *dx* and *dy*, resulting in 32 procedures, 16 each for horizontal and vertical translations. Half pixel values were ignored, and each procedure uses fixed point arithmetic with a 22 bit fractional part. In addition, for product term *T[i,j]*F[i]* in the generated code, we approximated the fixed point coefficient *T[i,j]* by zeroing the lower eight bits, which allowed the compiler to replace the multiply with several shifts and adds.

However, my initial experiments showed that dynamic thresholding, by itself, did not provide either sufficient speedup or satisfactory image quality. It was necessary to use caching (section 4.2) and temporary requantization (section 4.3) to improve speed and quality further.

6.4.3 Performance Optimization II - Caching

Caching is a technique that reduces the number of translation computations required. Consider the process of computing two adjacent blocks $\mathbf{B_0}$ and $\mathbf{B_1}$ from reference blocks $\mathbf{F_0}$...$\mathbf{F_5}$, as shown in figure 6.6. $\mathbf{B_0}$ and $\mathbf{B_1}$ can be derived as follows:

$$\mathbf{G_0} = \mathbf{T_{dx}F_0} + \mathbf{T_{dx\text{-}8}F_1}$$

$$\mathbf{G_1} = \mathbf{T_{dx}F_2} + \mathbf{T_{dx\text{-}8}F_3}$$

$$\mathbf{G_2} = \mathbf{T_{dx}F_4} + \mathbf{T_{dx\text{-}8}F_5}$$

$$\mathbf{B_0} = \mathbf{T_{dy}G_0} + \mathbf{T_{dy\text{-}8}G_1}$$

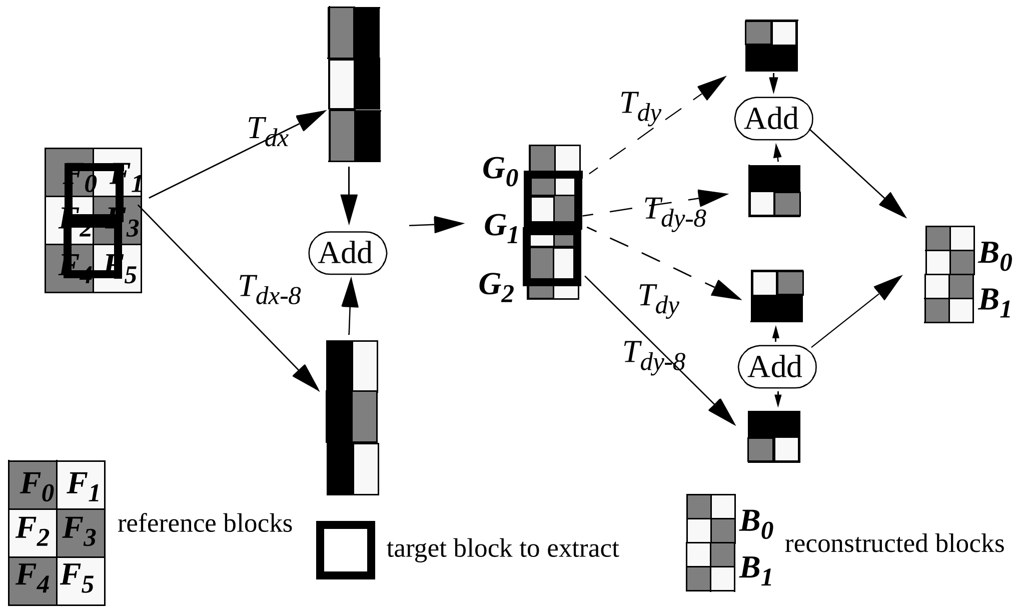


Figure 6.6: Detailed Predicted Vector Extraction Example

$$\mathbf{B_1 = T_{dy}G_1 + T_{dy\text{-}8}G_2}$$

Since the intermediate block $\mathbf{G_1}$ is required in the computation of both $\mathbf{B_0}$ and $\mathbf{B_1}$, computing $\mathbf{G_1}$ once for both $\mathbf{B_0}$ and $\mathbf{B_1}$ is advantageous. By doing so, it is possible to save a total of 4 out of 24 translation calculations per macroblock.

I generalized this idea further by caching the result of *every* translation of a reference block, since P or B blocks in other frames (or in the same frame but adjacent macroblocks) might require the reference block to be translated by exactly the same offset. I used a directly associative cache model: each reference block had dedicated cache entries for every potential translation in the x and y direction. This required a space allocation of 16 additional blocks per reference block. However, actual memory requirements were more reasonable because of the following factors:

- typically, less than half the frames in an MPEG sequence are reference frames.
- the mpeg decoder only decodes a small subset of all available frames at a time. Therefore, it allocates sufficient space in its data structures only for that small subset. Hence, allocating extra space for caching would not significantly augment the transcoder primary memory requirements.

6.4.4 Quality Optimization - Temporary Requantization

Dynamic thresholding is only advantageous when SC blocks are sparse (i.e., *n* in equation 22 is small). To achieve low coefficient density, we initially quantized the input reference SC blocks (e.g. $\mathbf{F_0..F_5}$ in equation 6) using the standard MPEG quantization table, *prior* to performing the translation operations. Unfortunately, we found that this table scaled the coefficients too coarsely resulting in loss of detail in the final image. Hence, I used an intermediate table with smaller values[4] to quantize the input SC blocks. This resulted in sparser intermediate translated blocks $\mathbf{T_{dx}F}$ and $\mathbf{T_{dx}G}$ and lower transcoding speed. The performance drop, however, was offset by a significant improvement in final image quality.

6.5 Transcoding Experiments

To compare their performance, I implemented all 3 methods (spatial domain methods I and II and compressed domain method III). I used the Berkeley MPEG decoder and Independent JPEG group encoder to build the transcoders. Method I was straightforward to implement, methods II and III were more involved. I modified the decoder to store sparse 8x8 blocks as *RLE vectors*. Each RLE vector has an array of *(index, value)* pairs and a field indicating the size of the array. The translation transforms for the frequency domain approach converted RLE vectors to RLE vectors via procedures produced using a code-generator. Our transcoder is publicly available [67].

All experiments were run on an HP 735. I measured application performance by the number of frames transcoded per second. All data was read from, and written to, memory, so I/O was not a factor.

The quality of the output was measured using the PSNR metric:

$$PSNR = 20\log\left[\frac{255}{\sqrt{\frac{mse(C-O)}{imsize}}}\right]$$

[4] I used the MPEG quantization table scaled by 2

where *imsize* is the image size, *mse* is the mean square error function of an image compared to a reference image, *C* is the image produced by the transcoder and *O* is the corresponding image produced by decoding the MPEG stream to a sequence of gray-scale images.

I tested all these methods on a selection of MPEG streams whose properties are summarized in table 6.2.

Table 6.2: Properties of the test MPEG sequences

Clip Name	Frame Size	Avg I (bytes)	Avg. P size	Avg. B size	Frame Pattern	Viewing fps
alesi.mpg	240x192	5043	4418	2243	IBBPBBPBBPBB	3
bike.mpg	352x240	11756	7106	1606	IBBPBB	5
bus.mpg	352x240	13512	7657	2723	IBBPBBPBBPBB PBB	4
cannon.mpg	192x144	7792	5918	1381	IBBPBB	5
us.mpg	352x240	6479	4660	1465	IBBPBB	5
raiders.mpg	160x128	1953			I	9

6.6 Results

Table 6.3 shows the transcoding performance and image quality (PSNR) for methods I, II, and III on a typical stream (“bike.mpg”). As we can see from the table, the PSNR for I-frames remains constant, regardless of method used. This is expected since the processing of I frames is almost identical in all methods. Method 1 gives the best picture quality but is also the slowest. Method II provides almost the same quality, but the speedup from method II is usually not significant. Method III provides significant performance improvement at the cost of slightly degraded picture quality, depending on the setting of *maxerr*. Performance leveled out for values of *maxerr* larger than about 15. Although method III avoids performing many operations in the translation procedures for large *maxerr*, the overhead of entropy encoding and decoding limit the speedup obtained. Profiling reveals about 50% of the code is spent in the translation procedures, with the remaining time spent in other procedures. This leads to the conclusion that the maximum speedup possible through compressed domain transcoding is about 3.6, and is limited by Huffman coding. It is interesting to note that the PSNR of the B frames is higher than that of the P frames. This property is due to the averaging techniques used in B block reconstruction, which act as a filter that reduces errors.

Table 6.3: Speed and Quality of Various Transcoders On "bike.mpg" sequence

Method	**frames/sec**	**PSNR for I**	**PSNR for P**	**PSNR for B**	**Speedup**
Method I	5.7	36.3	37.5	37.8	1.0
Method II	5.9	36.3	37.4	37.7	1.0
Method III	6.5	36.3	35.3	35.7	1.1
maxerr=0					
maxerr=5	8.1	36.3	35.0	35.3	1.4
maxerr=10	9.6	36.3	34.5	34.7	1.7
maxerr=15	10.5	36.3	33.5	33.6	1.8

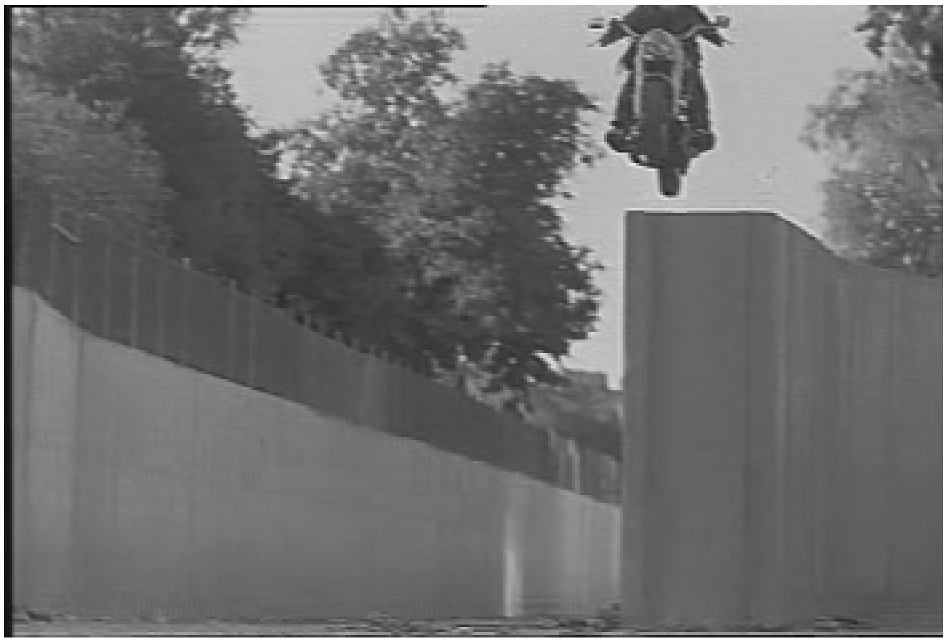


Figure 6.7: B frame decoded directly to gray-scale

The image quality of MPEG sequences with a long frame pattern of P and B frames between successive I frames (such as “alesi.mpg”) tend to degrade on the order of 0.1-1dB between successive P and B frames *within* the group of pictures. This behavior is due to the error generation in the transcoding of a P frame which then propagates to subsequent P and B frames. Long frame sequences also impair transcoding performance because of the presence of proportionately more non-reference frames and, thus, the need for more translation operations necessary to restore these frames.

Figure 6.7 shows a sample B frame from “bus.mpg”. Figures 6.8 and 6.9 present the same still but generated via method III with *maxerr* 0 and 10 respectively. All three images are available on-line [69].

Table 6.4 compares the performance of all three techniques on a variety of streams. These measurements show that the speedup from method II is usually small. The compressed domain approach typically improves performance

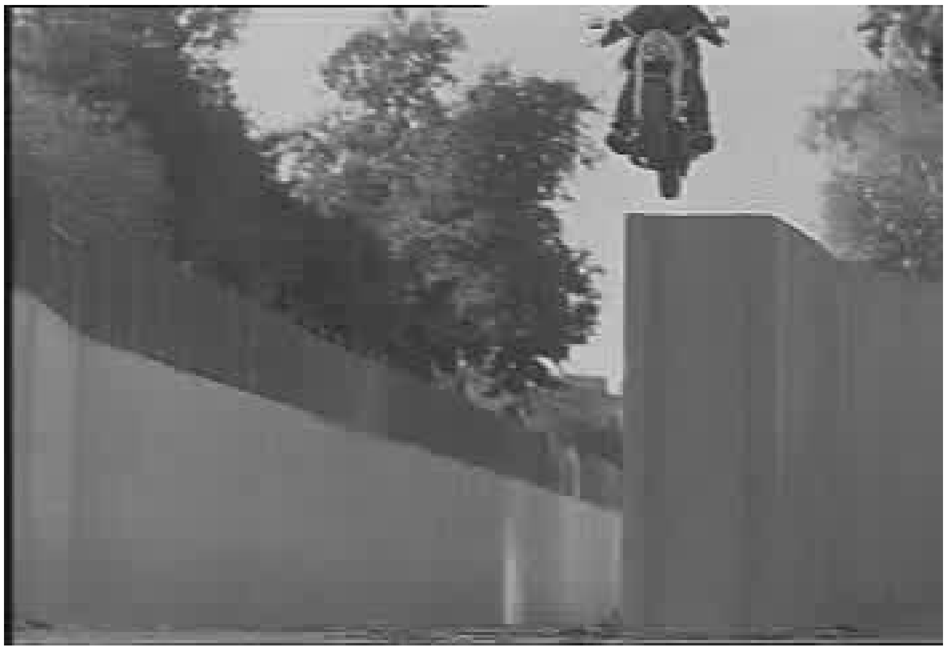


Figure 6.8: Same frame decoded with *maxerr*=0, PSNR = 35.7

by 1.5 to 3 times. Notable exceptions are “bus.mpg” and “raiders.mpg”. The former is encoded via intra- and non intra-block quantization tables that are not the default values recommended by the official MPEG-1 standard. Since the translation procedures are optimized for the standard tables, both transcoding performance and image quality suffer. The long frame pattern length of “bus.mpg” is also another reason for its poor transcoding performance. “Raiders.mpg” consists of nothing but I-frames, which explains why the frame rate of

Table 6.4: Speed of Various Transcoders (frames transcoded/second)

MPEG Clip	**Method I**	**Method II**	**Method III maxerr=0**	**Method III maxerr=1**	**Method III maxerr=5**	**Method III maxerr=10**
alesi.mpg	9.7	11.2	9.7	10.0	11.8	14.3
bike.mpg	5.7	5.9	6.5	6.6	8.1	9.6
bus.mpg	5.3	5.1	3.8	3.8	4.4	5.4
cannon.mpg	15.7	16.1	28.0	28.6	31.4	34.4
us.mpg	6.0	6.3	13.6	14.1	15.9	17.1
raiders.mpg	20.5	90.0	143.8	143.8	143.8	143.8

“raiders.mpg” remains the same for all values of *maxerr* in method III. The

---

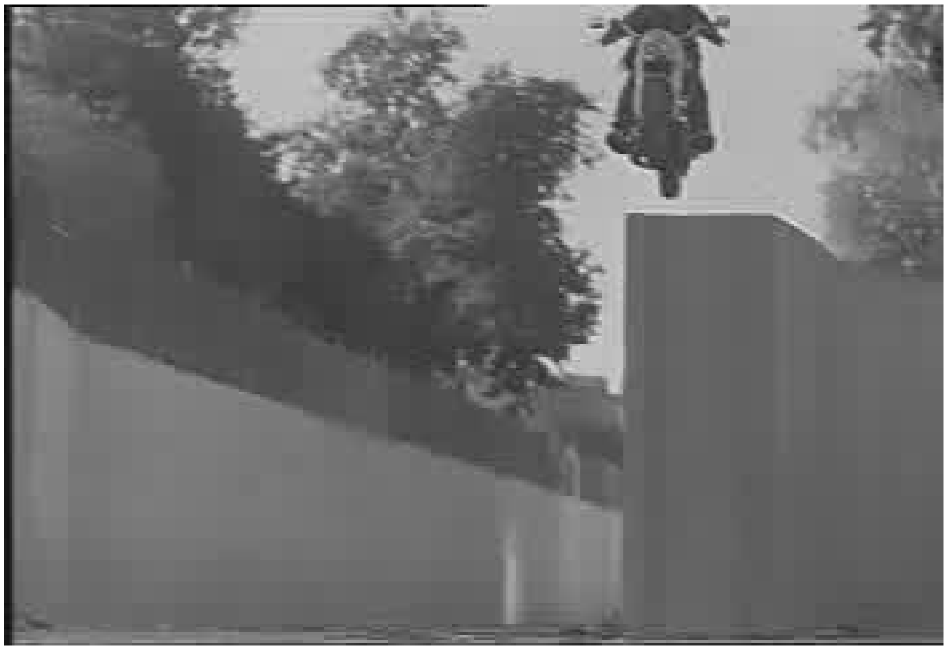


Figure 6.9: Same frame decoded with *maxerr*=10, PSNR = 33.6

dynamic thresholding routines only work for P and B motion compensated frames.

Table 6.4 also shows that despite being similar in picture size, there is a noticeable performance difference between “bike.mpg” and “us.mpg”. This is explained by the frame sizes of the MPEG bitstreams. On average, each compressed I frame from “bike.mpg” is 11756 bytes in size. P and B frames are 7106 and 1606 bytes respectively. In contrast, each I frame from “us.mpg” is 6479 bytes. P and B frames are 4660 and 1465 bytes. The smaller data sizes for “us.mpg” implies sparser blocks resulting in a greater speedup for the compressed domain processing. Another reason is that “bike.mpg” has a longer frame pattern than “us.mpg”. Hence, the transcoder has to perform relatively more block extraction operations in the former sequence than the latter resulting in poorer performance for “bike.mpg”.

Caching saved about 37%-40% of all potential translation operations in all our test streams except “cannon.mpg” where a high cache hit rate of 46% and a small frame sequence greatly offset the transcoding costs of a stream with a relatively high bit rate.

Overall, the performance of compressed domain transcoding depends on the bitrate of the streams, the length of the frame pattern and the degree of caching achieved.

6.7 Related Work

Chang and Messerschmitt [13, 14, 15] developed the first techniques for solving the macroblock alignment problem in the compressed domain. Their approach is to pre- and post-multiply the reference blocks by appropriate matrices. By using the distributive property of matrix multiplication with respect to the DCT , they are able to pre-compute the pre- and post matrices for a range of effects in the compressed domain. They report on the performance of software simulations for video compositing in the compressed domain for motion compensated video in [15], a process that involves block translation as well as scaling, compositing and motion vector search, all in the compressed domain. Additionally, they mention the need for intermediate quantization to preserve image quality. Keesman et al provide an overview of the design space and describe a generalized architecture for MPEG transcoding in [28].

Merhav and Bhaskaran [36, 37, 38] propose optimizations to the algorithms presented by Chang and Messerschmitt. In particular, they concentrate on the scaling (downsampling) and source block extraction (they refer to it as *inverse motion compensation*) techniques. They observe that the original methods work best if the input DCT blocks are sparse and the degree of translation for a large fraction of the reference blocks are zero in either *dx* or *dy*. Their improvements are largely based on effective factorization of the pre- and post-matrices as well as the input blocks.

Sethi and Shen [47] examine the special case of inner block transforms (IBTs), which are form of factoring compressed domain operations. They utilize the computational symmetry present in the calculations of certain image operations in the DCT domain to simplify their IBTs. IBTs, together with pixel addition, can be used to implement a wide variety of image operations. They also [48] examine the decomposition of complex affine transforms, such as shearing and perspective mapping, into simpler multipass operations. Our choice to implement the translation operation as a sequence of translations in the x and y access follows similar logic.

In contrast to the first three approaches, the work presented in this chapter explores more of the design space by comparing spatial and compressed domain transcoding, and is more systems-oriented. This led to insights that would not have been possible by a theoretical-only treatment of the problem. For example, I found that the computational complexity involved in the translation operations was not the only major bottleneck to transcoding. Entropy encoding and decoding and conversion of the translated matrix to run length vectors also accounted for significant computational costs.

Other compressed domain processing techniques typically focus on specialized (but important) problems. Yeo and Liu have studied the problem of cut-detection on MPEG data [62]. They approximate Chang's techniques to extract the DC values from a compressed video stream [63] and build a sequence of DC images. They use the DC images to compute various forms of scene changes between successive frames via a collection of threshold-based metrics. Natarajan and Bhaskaran [40] show that the operation of shrinking an image by a factor of two can be implemented in the compressed domain using only shifts and adds. Their method approximates the transform matrices as powers of 2. Sethi and Shen [48] have examined the role of convolution in the DCT domain and used their techniques to implement compressed domain edge detection. They employ the post- and pre- matrix multiplication technique to absorb the computation of the convolution masks in the DCT domain. Further savings are possible if the convolution mask is symmetric.

Content-based search on images and video is another related area of research. Seales [46] has done work on object recognition using an eigenspace method in the compressed domain, and Arman et al [5, 6] have shown how to perform cut detection in motion-JPEG video data. They rely on a thresholding metric computed from the inner product of two vectors formed from the coefficients of the DCT blocks of the frames under consideration. The metric reduces the number of frames that have to decompressed to the spatial domain for more detailed analysis.

6.8 Conclusions

In this chapter I have evaluated the performance of software implementations of compressed domain processing for the problem of MPEG to JPEG transcoding.

The method of implementing compressed domain processing is based on code generation and dynamic thresholding. Transcoding speed is proportional to the bitrate of the compressed video stream, the length of the frame pattern and the degree of caching possible. Transcoding speed can also be dynamically varied, a useful feature in real-time systems where time is critical.

The MPEG to JPEG transcoding speed obtained by almost any imaginable compressed domain technique is ultimately limited by the entropy coding methods used in MPEG and JPEG, which are not well suited for software implementation. This observation applies to other types of compressed domain processing as well. Perhaps we may conclude from this that next generation compression standards should consider alternative entropy coding techniques that are more well suited for software implementations. Given the complexity of compression standard like MPEG-4, such a change would cost very little in complexity, but improve our ability to process the compressed data.

Chapter 7

Future Work and Conclusions

This section enumerates the specific contributions of this thesis in the domain of multimedia communications and indicates directions for extending this work.

7.1 Video Surveys

The first contribution of this thesis are two surveys that investigate the characteristics of videos on the web (chapter 2) and how videos are accessed (chapter 3). In chapter 2, I developed a suite of tools that detected, downloaded, and analyzed about 57000 video files on the web -- about one hundred GB of data. My findings included:

1. *Web video size:* Videos are around 1 Mbytes in size, an order of magnitude larger than HTML documents which are usually sized around 1-2 Kbytes. Playback time is about a minute or less.
2. *WORM nature:* Web videos tend to follow the write-once-read-many principle. Once a video has been placed online, chances are that it will stay there. Hence, cache consistency is not a major issue in video caching systems.
3. *High bandwidth requirements:* A high percentage of web video material cannot be downloaded and played back in real-time as current network/modem bandwidths are not enough to meet their implicit playback requirements.

4. *Growth*: the number of videos on the web are growing at an almost exponential rate.

The results in chapter 2 indicate a number of directions for future work. A followup study with a broader scope, for example, could be used to confirm the general video characteristics uncovered in the first study. The second study could also clarify the reason behind the drop in the number of movies coming online. Such a study would also provide an opportunity to inspect the popularity and content of streaming videos.

In chapter 3, I analyzed access logs to a video server for a VOW experiment in Lulea University in Sweden. The Lulea server stores 140 files containing seven movies and 133 recordings of class lectures and seminars. The analysis yielded the following insights:

1. *Future trends:* videos become larger when network bandwidth increases and low bitrate streaming protocols are used. The median size and duration of files at the Lulea University video server was 110 MBytes and 77 minutes. Lulea uses a high-bandwidth (34 megabits/second) network and H.261 based multicast architecture.
2. *Inter-arrival times*: the median interarrival time of requests for videos at Lulea was about 400 seconds. This indicates requests are infrequent compared to HTML documents.
3. *Video browsing patterns:* users like to view the initial part of videos in order to determine if they are interested or not. If they like it, they continue watching. Otherwise, they stop.

4. *Temporal Locality:* accesses to videos also exhibit strong temporal locality. If a video has been accessed recently, chances are that it will be accessed again soon.

Like chapter 2, chapter 3 would also benefit from further studies that corroborated its findings. In particular, two types of studies are necessary. The first type would repeat the chapter 3 analysis on the same server but for a longer duration of accesses. Given that the log data for the initial study ranged for six months, it would be interesting to see if similar patterns emerge from inspecting a year's worth of data. The second type of study would repeat the analysis on traces from other VOW projects to see if the same patterns emerge.

7.2 MiddleMan

The second contribution of this thesis is the architecture and analysis of MiddleMan, a video caching web proxy system. By caching videos close to clients and achieving high hit rates, MiddleMan is able to dramatically reduce overall access latencies. Typically, 60-90% of all off-campus requests can be served from the cache. Additionally, the caching algorithm used in the system is effectively able to balance load across multiple proxies and quickly adjust to sudden increases in the number of client requests. From the point of view of the server, MiddleMan dramatically reduces load by intercepting a large number of server accesses. Hence, the net effect of MiddleMan is to greatly increase the effective bandwidth of the entire video delivery system by a factor between three and ten, allowing more clients to be serviced at any given time.

MiddleMan shows promise, but raises a number of issues that need to be addressed. These include:

- *Fault tolerance:* the current architecture of MiddleMan renders it susceptible to proxy or coordinator crashes. In particular, a coordinator crash causes the system to become unusable since only the coordinator maintains global system state. Standard techniques such as reliable backup servers may provide a possible solution but leads to two further problems. First, proxies must be able to detect the location of the new coordinator. Second, the process of switching from one coordinator to another might stall ongoing proxy-client connections. One possible approach to the first problem is to maintain a separate multicast channel within the cluster which can be used to inform all proxies of any configuration changes. The second problem might be avoided by the proxy bypassing MiddleMan and fetching the next block directly from the WWW server.
- *Fast-forward/Rewind Support:* clients may wish to fast-forward or rewind through video material. Currently, MiddleMan does not explicitly support such functionality. However, both the proxies and the cache replacement policy can be altered so that once the proxy detects a fast-forward or rewind request from the client, it is able to request the right sequence of blocks from the coordinator.
- *Security/Authentication:* the proxy cache might contain "pay per view" type movies which, if not checked, might allow clients to access titles without authorization from the original movie provider. Hence, an authentication scheme is necessary which would allow MiddleMan to verify whether a client is allowed to retrieve a certain title from the cache. It might also be necessary to encrypt cache contents to prevent unauthorized access to files.
- *Proxy Cluster Cooperation:* chapters 4 and 5 have investigated the protocols and algorithms necessary for the functioning of a single proxy cluster. An

obvious next step would be to increase the scope of the system by allowing multiple proxy clusters to interact. In this scenario, if a file was not available in the local proxy, the coordinator might redirect the request to a proxy in a different cluster which does have the file cached. One possible method for achieving such cooperation is for coordinators to periodically exchange data about cache contents on some well known multicast channel.

Future work on MiddleMan will focus on addressing these issues as well as building and deploying a prototype.

7.3 Transcoding

Chapter 6 described a novel, high-performance technique for fast transcoding from MPEG to M-JPEG. The technique was based on the observation that the JPEG compression algorithm (and the compression scheme for MPEG I frames) can be decomposed into a linear transformation (e.g., DCT + zig-zag scan + constant division) followed by a lossy, non-linear step (e.g., integer rounding) followed by an entropy coding step (e.g., Huffman coding). By writing the calculation of a pixel translation as a linear transformation and combining it with the linear transformation of the JPEG algorithm, I developed a compressed domain translation operator. By exploiting the approximation introduced by the lossy step in JPEG (and part of MPEG), and by introducing a framework for selectively skipping multiplies, I developed a technique called dynamic condensation that reduced the complexity of compressed domain calculation in proportion to the computational error desired in the final output. I used this approach to implement a compressed domain translation operator, the core of my compressed domain transcoder. In general, I found the transcoder to be about 1.5 to 3 times faster than its spatial domain counterpart. Transcoding speed can also be dynamically varied at the expense of picture quality, a useful feature in real-time systems where time is critical.

Though the technology has promise, further work remains before it can be integrated into the MiddleMan infrastructure. In particular, the transcoder has to be combined with a proxy-client video delivery protocol that is able to detect client and local network loads and control transcoder output accordingly.

Additionally, techniques developed for this transcoder, such as the compressed domain translation and dynamic thresholding, can be utilized in a JPEG to MPEG transcoder. Such a transcoder could be used for off-line recompression of proxy contents since MPEG typically is more efficient at video storage than JPEG. Additionally, this transcoder could also be used to adapt high bitrate videos to low bandwidth proxy-client connections.

Appendix A

Computing the J and $\bar{J}$ operators

```
#define U       0
#define V       1

// Lookup table to encode the zig-zag scan order
static int zz[64][2] = {
    {1, 1},
    {2, 1}, {1, 2},
    {1, 3}, {2, 2}, {3, 1},
    {4, 1}, {3, 2}, {2, 3}, {1, 4},
    {1, 5}, {2, 4}, {3, 3}, {4, 2}, {5, 1},
    {6, 1}, {5, 2}, {4, 3}, {3, 4}, {2, 5}, {1, 6},
    {1, 7}, {2, 6}, {3, 5}, {4, 4}, {5, 3}, {6, 2}, {7, 1},
    {8, 1}, {7, 2}, {6, 3}, {5, 4}, {4, 5}, {3, 6}, {2, 7}, {1, 8},
    {2, 8}, {3, 7}, {4, 6}, {5, 5}, {6, 4}, {7, 3}, {8, 2},
    {8, 3}, {7, 4}, {6, 5}, {5, 6}, {4, 7}, {3, 8},
    {4, 8}, {5, 7}, {6, 6}, {7, 5}, {8, 4},
    {8, 5}, {7, 6}, {6, 7}, {5, 8},
    {6, 8}, {7, 7}, {8, 6},
    {8, 7}, {7, 8},
    {8, 8}};

// The default quantization table
static double q[8][8] = {
    { 16,  11,  12,  14,  12,  10,  16,  14},
    { 13,  14,  18,  17,  16,  19,  24,  40},
    { 26,  24,  22,  22,  24,  49,  35,  37},
    { 29,  40,  58,  51,  61,  60,  57,  51},
    { 56,  55,  64,  72,  92,  78,  64,  68},
    { 87,  69,  55,  56,  80, 109,  81,  87},
    { 95,  98, 103, 104, 103,  62,  77, 113},
   {121, 112, 100, 120,  92, 101, 103,  99}};

// The following functions compute the compressed domain operators,
// as desribed by Smith []

#define A(u)    ((u)? 0.5 : 0.5/SQRT2)
double C(int i,u) {
    return A(u)*cos((2*i+1)*u*PI/16.0);
}

double J(int i,j,k) {
    int u = zz[k][U]-1;
    int v = zz[k][V]-1;
    return C(i,u)*C(j,v)/q[u][v];
}

double Jhat(int k,i,j) {
    int u = zz[k][U]-1;
    int v = zz[k][V]-1;
    return C(i,u)*C(j,v)*q[u][v];
}

{
    double sum, T[64][64];
      for (k=0; k<64; k++) {
            for (l=0; l<64; l++) {
                sum = 0.0;
                for (i=0; i<8; i++)
```

```
                    for (j=0; j<8-dx; j++)
                        sum += J(i, j+dx, l)*Jhat(k,i,j);
                T[k][l] = sum;
            }
}
```

Code To Compute $\mathbf{T}_{\mathbf{dx}}$

Bibliography

[1] M. Abrams et. al, *Caching Proxies: Limitations and Potentials*, 4th International World-Wide Web Conference, pp 119-133, Decemeber 1995

[2] C. C. Aggarwal, J. L. Wolf, P. S. Yu, *On Optimal Batching Policies for Video-on-Demand Storage Servers*, *Proc. ACM Multimedia'96, pp. 253-258, 1996*

[3] V. Almeida et al, *Characterizing Reference Locality in the WWW*, Technical Report TR-96-11, Department of Computer Science, Boston University, 1996

[4] E. Amir, S. McCanne, S. H. Zhang, *An Application-level Video Gateway*, Proc. of the Third ACM International Conference on Multimedia, November 1995, San Francisco, CA, pp. 511-522

[5] F. Arman, A. Hsu, M-Y. Chiu, *Image Processing on Compressed Data for Large Video Databases*, Proceedings of the First ACM International Conference on Multimedia, Anaheim, CA, August 1993

[6] F. Arman, A. Hsu, and M-Y. Chiu, *Image Processing on Encoded Video Sequences*, ACM Multimedia Systems Journal, 1994

[7] M. Arlitt, C. Williamson, *Web Server Workload Characterization: The Search for Invariants*, ACM SIGMETRICS 96-5, Philadelphia, PA, USA, 1996

[8] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, J. K. Ousterhout,

Measurements of a Distributed File System, in Proc. 13th SOSP, Oct. 1991

[9] A. Bestavros et al, *Application Level Document Caching in the Internet*, in Proceedings of Workshop on Services and Distributed Environments, June 1995

[10] T. Bray, *Measuring the Web*, in Proc. 4th International World Wide Web Conference, Paris, France, May 1996, pp 994-1005, <URL: http://-www5conf.inria.fr/fich_html/papers/P9/Overview.html>

[11] D. W. Brubeck, L. A. Rowe, *Hierarchical Storage Management in a Distributed VOD System*, IEEE MultiMedia, Fall 1996, Vol. 3, No. 3

[12] A. Castro, C. Lazzerini, V. Kolla, Massively Distributed Video File Server Simulation: Investigating Intelligent Caching Schemes, <URL: http://www.cs.cornell.edu/Info/Projects/zeno/DVFS/EdAlex/EdAlex.ps>

[13] S-F. Chang, W-L. Chen, D. G. Messerschmitt, *Video Compositing in the DCT Domain*, IEEE Workshop on Visual Signal Processing and Communications, Raleigh, NC, pp. 138-143, Sep. 1992

[14] S-F. Chang, D. G. Messerschmitt, *A New Approach to Decoding and Compositing Motion Compensated DCT-Based Images*, IEEE Intern. Conf. on Acoustics, Speech, and Signal Processing, Minneapolis, Minnesota, pp. V421-V424, April, 1993

[15] S-F. Chang, D. G. Messerschmitt, *Manipulation and Compositing of MC-DCT Compressed Video*, IEEE Journal of Selected Areas in Communications (JSAC), Special Issue on Intelligent Signal Processing, pp. 1-11, Jan. 1995

[16] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, K. Worrell, *A Hirerchical Internet Object Cache*, Proceedings of the 1996 USENIX Techni-

cal Conference, January 1996

[17] A. L. Chervenak, Tertiary Storage: An Evaluation of New Applications, Ph. D. Thesis, University of California at Berkeley, Computer Science Division Technical Report UDB/CSD 94/847, December, 1994

[18] C.R. Cunha, A. Bestavros, M. E. Crovella, *Characteristics of WWW Client Based Traces*, Technical Report TR-95-010, Computer Science Department, Boston University, July 1995

[19] A. Dan, D. Sitaram, P. Shahabuddin, *Scheduling Policies for an On-Demand Video Server with Batching*, Second Annual ACM Multimedia Conference and Exposition, San Francisco, CA, 1994

[20] H. Eriksson, *Mbone: The Multicast Backbone*, Communications of the ACM, Vol 8, pp 54-60, 1994

[21] W. Feng, S. Sechrest, *Critical Bandwidth Allocation for Delivery of Compressed Prerecorded Video*, Proc. of the IS&T/SPIE Symposium on Multimedia Computing and Networking, pp 234-242, Feb. 1995

[22] W. Feng, S. Sechrest, *Critical Bandwidth Allocation for Delivery of Compressed Video*, Computer Communication, vol. 18, pp. 709-717, Oct. 1995

[23] W. Feng, F. Jahanian, S. Sechrest, *Optimal Buffering for the Delivery of Compressed Prerecorded Video*, Proc. of the IASTED/ISMM International Conference on Networks, Jan 1995

[24] W. Feng, J. Rexford, *A Comparison of Bandwidth Smoothing Techniques for the Transmission of Prerecorded Compressed Video*, Proc. IEEE INFOCOM, pp. 58-66, April 1997

[25] D. Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, Volume 34, No 4, pp. 46-58, April 1991

[26] K. L. Gong, L.A. Rowe, *Parallel MPEG-1 Video Encoding*, 1994 Picture Coding Symposium, Sacramento, CA, September 1994

[27] *Information Technology--Generic Coding of Moving Pictures and Associated Audio,* ISO/IEC 13818-2 Committee Draft (MPEG-2)

[28] G. Keesman, R. Hellinghuizen, F. Hoeksema, G. Heideman, *Transcoding of MPEG Bitstreams,* Signal Processing: Image Comm., vol. 8, pp. 481-500, Sept. 1996

[29] S. Keshav, *REAL: A Network Simulator,* Technical Report 88/472, Department of EECS, UC Berkeley, 1988

[30] T. T. Kwan, R. E. McGrath, D. A. Reed, *User Access Patterns to NCSA's World Wide Web Server*, CS Tech Report UIUCDCS-R-95-1934, University of Illinois at Urbana-Champaign, February 1995.

[31] A. Luotonen, K. Altis, *World-wide Web Proxies*, Computer Networks and ISDN Systems 27(2). 1994.

[32] A. Luotonen, *Web Proxy Servers*, Prentice Hall, 1998.

[33] S. McCanne, V. Jacobson, M. Vetterli, *Receiver-driven Layered Multicast*, Proc. ACM SIGCOMM '96. Also in Computer Communication review, vol. 26, no. 4 (Oct. 1996): 117-130

[34] S. McCanne, V. Jacobson, *vic: a Flexible Framework For Packet Video*, Proceedings of ACM Multimedia '95, Nov 1995

[35] J. McManus, K. Ross, *Video on demand over ATM: Constant-rate transmission and transport*, in Proc. of IEEE Infocom, pp 1357-1362, Mar. 1996

[36] N. Merhav, V. Bhaskaran, *Fast Inverse Motion Compensation Algorithms for MPEG and for Partial DCT Information*, J. Visual Communication and Image Representation, Vol. 7, no. 4, pp. 395-410, December 1996

[37] N. Merhav, V. Bhaskaran, *Fast Algorithms for DCT-Domain Down-Sampling and Inverse Motion Compensation*, IEEE Trans. on Circuits and Systems for Video Technology, April 1997

[38] N. Merhav, V. Bhaskaran, *A Fast Algorithm For DCT-Domain Inverse Motion Compensation*, *Proceedings of ICASSP 1996*, Volume IV, pp 2307-2310

[39] J. C. Mogul, *Network Behavior of a Busy Web Server and its Clients*, DEC WRL Research Report 95/5, October 1995

[40] B. Natarajan, V. Bhaskaran, *A Fast Approximate Algorithm for Scaling Down Digital Images in the DCT Domain*, IEEE International Conference on Image Processing, Oct. 23-26, 1995, Washington D.C.

[41] E. O'Neil, P. O'Neil, G. Weikum, T*he LRU-k Page Replacement Algorithm For Database Disk Buffering,* Proceedings of International Conference on Management of Data, May, 1993

[42] T. D.C. Little, D. Venkatesh, *Probabilistic Assignment of Movie to Storage Devices in a Video-On-Demand System*, Network and Operating System Support for Digital Audio and Video: Fourth International Workshop, Lancaster, UK, November, 1993. Springer-Verlag, Berlin; New York pp. 213-224

[43] J. Ousterhout, H.L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, *A Trace-Driven Analysis of the Unix 4.2 BSD File System*, Proc. 10th SOSP, December 1985

[44] P. Parnes , M. Mattsson, K. Synnes, D. Schefstrom, *mMOD - The multicast Media-On-Demand system*, extended abstract, January, 1997. <URL: http://www.cdt.luth.se/~peppar/docs/mMOD97/mMOD.ps>

[45] P. Parnes , K. Synnes, D. Schefstrom, *The CDT mStar Environment: Distributed Collaborative Teamwork in Action*, Third IT-conference in the Barit region, September 16-17, 1997, Luleå, Sweden

[46] W. B. Seales, M. D. Cutts, *Vision and Multimedia: Object Recognition in the Compressed Domain*, University of Kentucky Technical Report, Oct 1995

[47] B. Shen, I. K. Sethi, *Inner-block Operations on Compressed Images*, Proc. of the Third ACM International Conference on Multimedia, November 1995, San Francisco, CA, 1995.

[48] B. Shen, I. K. Sethi, *Scanline Algorithms in the JPEG Discrete Cosine Transform Compressed Domain*, Journal of Electronic Imaging, pp 182-190, April, 1996

[49] B. Shen, I. K. Sethi, V. Bhaskaran, *Digital Video Blue Screen Editing in Compressed Domain*, SPIE Conference of Visual Communications and Image Processing, San Jose, California, Jan. 28 - Feb. 2, 1997

[50] B. Shen, I. K. Sethi, V. Bhaskaran, *Closed-loop MPEG Video Rendering*, IEEE Conference on Multimedia Computing and Systems, Ottawa, Canada, June 1997

[51] B. Shen, I. K. Sethi, V. Bhaskaran, *DCT Convolution and its Application In Compressed Video Editing*, SPIE Conference of Visual Communications and Image Processing, San Jose, California, Jan. 28 - Feb. 2, 1997

[52] B. Shen, I. K. Sethi, *Convolution-Based Edge Detection for Image/Video in Block DCT Domain*, Journal of Visual Communication and Image Representation (In Press)

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

[54] B. C. Smith, L. A. Rowe, *Algorithms for Manipulating Compressed Images*, IEEE Computer Graphics and Applications, Volume 13, No 5, pp 34-42, Sept. 1993

[55] B. C. Smith, L. A. Rowe, *Compressed Domain Processing of JPEG-Encoded Images*, Real-Time Imaging, Volume 1, number 2, July, 1996, pp. 3-17

[56] J. R. Smith, S.-F. Chang, *Searching for Images and Videos on the World-Wide Web*, Columbia University CTR Technical Report 459-96-25, August 1996.

[57] K. Synnes, S. Lachapelle, P. Parnes, D. Schefstrom, *Distributed Education using the mStar Environment*, Research Report 1997:25, Luleå University of Technology, Sweden, December 01, 1997.

[58] R. Tewari, H.M. Vin, A. Dan, D. Sitaram, *Resource-based Caching for Web Servers*, Proceedings of ACM/SPIE Multimedia Computing and Networking 1998 (MMCN'98), San Jose, Pages 191-204, January 1998

[59] *Video Codec for audiovisual services at p*64 kbps*, 1993, ITU-T Recommendation H.261

[60] G. K. Wallace, *The JPEG Still Picture Compression Standard*, Communications of the ACM, Volume 34, No 4, pp. 30-44, April 1991

[61] A. Woodruff, P. M. Aoki, E. Brewer, P. Gauthier, L. A. Rowe, *An Investigation of Documents from the World Wide Web*, in Proc. 5th International World Wide Web Conference, Paris, France, May 1996, pp 963--979, <URL: http://www5conf.inria.fr/fich_html/papers/P7/-Overview.html>

[62] B-L. Yeo, B. Liu, *Rapid Scene Analysis on Compressed Video*, IEEE Transactions on Circuit and Systems for Video Technology, Vol. 5, No. 6, pp.

533-544, Dec. 1995

[63] B-L. Yeo, B. Liu, *On the extraction of dc sequence from mpeg compressed video*, IEEE International Conference on Image Processing, 1995.

[64] L. Zhang, S. Deering et al, *RSVP: A New Resource Reservation Protocol*, IEEE Network Magazine, Vol. 7, No. 5, Sep 1993, pp 8-19

[65] G. Zipf, *Human Behaviour and the Principle of Least Effort,* Addison-Wesley, 1949.

[66] ftp://bmrc.berkeley.edu/pub/mpeg/stat/

[67] ftp://ftp.cs.cornell.edu/pub/multimed/m2j.tar.gz

[68] D. Wessels, “Squid Internet Object Cache”, http://squid.nlanr.net/Squid

[69] http://www.cs.cornell.edu/Info/Projects/zeno/Papers/Transcode/index.html

[70] http://www.altavista.digital.com/

[71] http://www.sunscript.com

[72] http://www.rahul.net/jfm/avi.html

[73] http://quicktime.apple.com

[74] http://www.cs.cornell.edu/Info/Projects/zeno/Projects/Tcl-DP.html

[75] http://xanim.va.pubnix.com/home.html

[76] http://www.vxtreme.com

[77] http://www.vdolive.com

[78] http://cu-seeme.cornell.edu/

[79] http://www.realmedia.com

[80] http://www.jaggedinternetworks.com/

[81] http://www.scour.net/

[82] http://www.zdnet.com/zdnn/stories/news/0,4586,2139668,00.html

[83] http://www.streamland.com/

[84] http://www.broadcast.com/

[85] http://www.freespeech.org/

[86] http://www.cc.gatech.edu/fce/c2000/

[87] http://www.cs.cornell.edu/zeno/Projects/lecture%20browser/Default.html

[88] http://scpd.stanford.edu:8000/

[89] http://home.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html

[90] http://developer.netscape.com/docs/manuals/communic-
tor/jsguide/js1_2.htm

[91] http://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-rev-06.txt

[92] http://squid.nlanr.net/squid

[93] http://www.w3.org/Jigsaw/

[94] http://java.sun.com/

[95] http://www.rahul.net/jfm/mpeg4.htm

[96] http://www.internic.net

[97] http://cdt.luth.se/

[98] http://mmod.cdt.luth.se/