STREAMOBJECTS: DYNAMICALLY-SEGMENTED SCALABLE MEDIA OVER THE INTERNET

by

STEVEN NIEMCZYK

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 27, 1996

Author [illegible]
Department of Electrical Engineering and Computer Science
May 27, 1996

Certified by [illegible]
Steven R. Lerman
Thesis Supervisor

Accepted by [illegible]
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

To Mom and Dad, for everything

StreamObjects: Dynamically-Segmented Scalable Media Over the Internet

by

Steven Niemczyk

Submitted to the Department of Electrical Engineering and Computer Science

May 27, 1996

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The proposed StreamObjects system provides a method of distributing dynamically-segmented, scalable multimedia over the Internet using the conventional client/server technologies used on the World Wide Web.

The system is designed to integrate directly with a standard Web server to provide new delivery modes and greater encoding flexibility. Servers could now encode a video or audio program in its entirety and create segments of the show on demand, lowering labor and resource requirements. A server could also provide multiple qualities and formats from a single source file, adapting to the diverse bandwidth and platform constraints of the Internet.

By providing low latency access to only the desired part of a media clip, tremendous gains are achieved in the area of network and server efficiency. The success of the StreamObjects system will continue to grow with future projects, potentially impacting the way multimedia is delivered over the Internet.

Thesis Supervisor: Dr. Steven R. Lerman
Title: Professor of Civil and Environmental Engineering; Director of the Center for Educational Computing Initiatives

ACKNOWLEDGMENTS

I would like to acknowledge Professor Steven Lerman, for all his ideas, support, and guidance, in helping make this project a reality. I would like to acknowledge Philip Bailey and Thomas Lee, for their guidance, support, and inspiration throughout this project. I would like to acknowledge the Center for Advanced Education Services and the Networked Multimedia Information Services project for providing the funding and resources critical to the success of the project.

I would also like to acknowledge the Student Information Processing Board and Chris Provenzano for their help with his wonderful thread package and their sound advice and support with UNIX development in general. I would like to acknowledge my friends Jason Hintersteiner and Erik Nygren for advice, tips, pointers and guidance throughout this endeavor.

I would also like to thank my countless friends at Baker House for support, encouragement, happiness, and diversion, giving me perspective that kept me focused on the task at hand. Most of all, I would like to acknowledge my parents, for everything, without which none of this could be possible.

TABLE OF CONTENTS

<u>TABLE OF FIGURES</u>

TABLE OF TABLES

List of Abbreviations

API	Application Programming Interface
AU	Access Unit
CD-ROM	Compact Disc Read Only Memory
CERN	Center for European Nuclear Research[1]
CGI	Common Gateway Interface
CMU	Carnegie Mellon University
CNN	Cable News Network
CPU	Central Processing Unit
DCE	Distributed Computing Environment
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISDN	Integrated Services Digital Network
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
I/O	Input / Output
ISO	International Standards Organization
MIT	Massachusetts Institute of Technology
MPEG	Moving Pictures Expert Group
NMIS	Networked Multimedia and Information Services
NCSA	National Center for Supercomputing Applications
OM-1	Open MPEG-1
PU	Presentation Unit
RTTI	Runtime Type Identification
SIPB	Student Information Processing Board
TESI	Turner Educational Services Incorporated
UC	University of California
VCR	Video Cassette Recorder
WWW	World Wide Web

[1] The acronym for CERN is in French, explaining the reversal of the N and the R.

CHAPTER 1: INTRODUCTION

1.1 OBJECTIVES

The StreamObjects system provides a method of distributing dynamically-segmented, scalable multimedia over the Internet using the conventional client / server technologies of the World Wide Web (WWW). With the StreamObjects system, standard Web servers can allow greater flexibility and control in determining what kinds of content can be delivered over the Internet. With the StreamObjects system in place, a client can download a dynamically created media stream containing only the precise segment of video and/or audio desired, without creating another (potentially large) file on the server, and without requiring expensive specialized media servers.

With the StreamObjects system, clients can choose which tracks to download among the available audio or video tracks stored within a single master media file. This dynamic approach greatly benefits both client and server, reducing the load on both sides over more traditional techniques.

The StreamObjects system, designed and implemented by the author, is based on a simple, portable, and extensible design. This enables the system to be deployed on platforms previously unforeseen, and to utilize and interact with protocols and interfaces beyond that of the World Wide Web and the Internet. This master's thesis outlines the StreamObjects project, its applications, how it was designed and implemented by the author, its performance, and its future in a world of ever changing information technologies.

1.2 Motivation

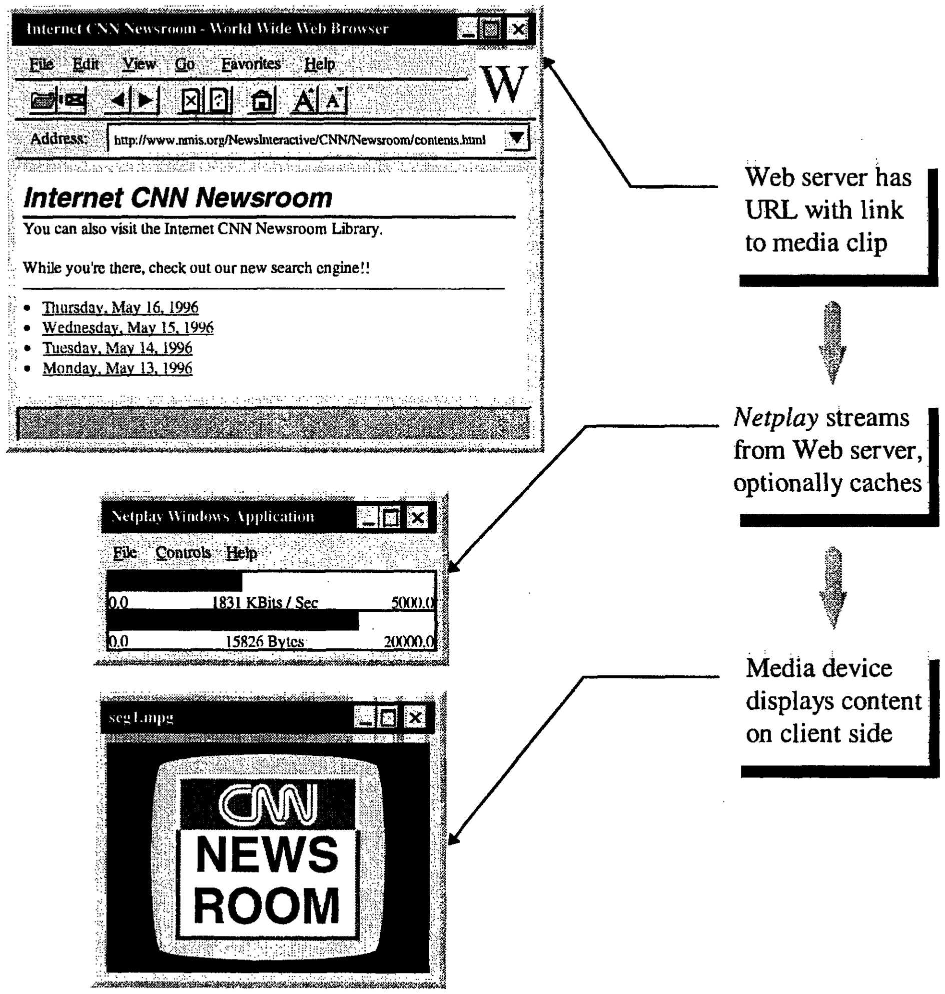
There are several motivations for creating the StreamObjects system. Currently, there is no standard method of streaming multimedia files^2. Without such a standard, the World Wide Web and its associated Hypertext Transfer Protocol (HTTP) is commonly used. Because HTTP is a stateless protocol, interactive media delivery is not done – the more traditional method of "download, store, and playback" is used.

This method has several disadvantages. First, it has high latency, since one must wait for the entire file to download. Second, it requires vast amounts of temporary storage on the client side, even if the file will only be viewed once, or if only a portion of the file will be viewed. Third, there is also the potential for significant wasted bandwidth, especially in the case where only a small portion of the downloaded media file is actually needed for viewing or archiving.

The *Netplay* streamer, developed by Jonathan Soo and later enhanced by the author, both of the Network Multimedia Information Services (NMIS) project, overcame some of the inadequacies of the HTTP download-and-playback model, by providing a method for streaming media files directly to certain MPEG decoders in parallel with the download phase. This solves the latency and storage issues, but does not eliminate the wasted bandwidth concerns. In addition, the Netplay streaming solution provides no method of streaming from a particular time within a media file. As described below, the StreamObjects system provides a solution that, in conjunction with the Netplay streamer or without, provides a more general media delivery strategy (see Figure 1).

Another motivation for the StreamObjects system originated from another major part of the NMIS group – the *Internet CNN Newsroom* project. This project, formerly coordinated by Kip Compton, and currently by Douglas Schreiber, automatically encodes *CNN Newsroom*, a educational program broadcast on CNN as part of the "Cable In the Classroom" initiative.

CNN Newsroom each weekday consists of several news stories in current events that *Internet CNN Newsroom* digitally encodes automatically each day. Time codes marking the beginning of each segment of the show are sent via electronic mail to NMIS each day by Turner Educational Services Incorporated (TESI), hours before broadcast of the show on CNN.


Internet CNN Newsroom - World Wide Web Browser
File Edit View Go Favorites Help
Address: http://www.nmis.org/NewsInteractive/CNN/Newsroom/contents.html
Web server has URL with link to media clip
Netplay streams from Web server, optionally caches
Netplay Windows Application
File Controls Help
0.0 1831 KBits / Sec 5000.0
0.0 15826 Bytes 20000.0
Media device displays content on client side
seg1.mpg
CNN NEWS ROOM

Internet CNN Newsroom

You can also visit the Internet CNN Newsroom Library.

While you're there, check out our new search engine!!

- Thursday, May 16, 1996
- Wednesday, May 15, 1996
- Tuesday, May 14, 1996
- Monday, May 13, 1996

Figure 1: Netplay Streaming Paradigm for the World Wide Web

A professional quality Beta[3] deck is set to record the show at its broadcast time in the early hours of the morning. An automated script controls both the VCR and the MPEG encoding station, running a set of programs digitally encodes and stores each segment of the show

---

[2] Several potential standards have been proposed at the time of this writing, including *ActiveMovie* by Microsoft Corporation, and *LiveVideo* by Netscape.

[3] The high-quality Beta analog video format, developed by the Sony Corporation, is widely used commercially for video production and editing.

separately, using the time codes given for that day[4]. This is necessary because there is no *in-band* indicator when a segment begins or ends. Each file (seg1, seg2, etc.) is transferred from the encoding station to our web server, where the web pages are automatically generated for each day's content using a guide created by *Teachable Text*, for TESI. This guide contains story descriptions and questions for use by the students who use *CNN Newsroom*, or our Internet equivalent.

With some frequency, the time codes for a show will be incorrect, often with each segment's time code off by a fixed constant. This requires our group to re-encode the video for the day since the existing video files would all be incorrect because the borders between segments would be incorrect.

This is a time-consuming and taxing operation, requiring an operator to reacquire the correct time codes, regenerate a script for encoding the video, and start the encoding process again, which takes over an hour to complete. In a professional, commercial-grade production system[5], these problems would be costly, as well as time-consuming.

The StreamObjects system offers a quicker, cheaper, simpler solution to this problem. Since the StreamObjects system can automatically generate a sub-section of a large media file *dynamically*, the need to encode each segment of the show separately is eliminated. The complex encoding script could be simplified to encode the show in its entirety, leaving slack in the start and stop times to handle any small variations in the broadcast time.

Once the show is taped and digitally encoded (which could now be combined into one pass), the single file could be dynamically segmented whenever a user requests a segment within the show. The StreamObjects system would move the segmentation from the encoding station's script to the Hypertext Markup Language (HTML) pages that contain the links to each segment.

[4] Time codes are received daily from TESI and indicate when a segment ends and another begins.

Adjusting the time codes would simply mean adjusting links on the web pages, which could be done by hand, or automatically by a script in seconds. Re-encoding would become a rare occurrence, only for the most egregious of errors in the encoding process, which may not be recoverable at all.

In addition to simplifying the encoding process, the StreamObjects system allows the *Internet CNN Newsroom* program, or any other content provider using the system, the ability to provide newer, more flexible qualities of service, *without* re-encoding the existing video. With the StreamObjects system, each segment could be provided with video only, audio only, or the standard synchronized video/audio delivery. Support for "teaser" files could also be provided containing only a few seconds of content to determine if the user wishes to download the larger, actual segment.

Lower bandwidth streaming could be provided for clients with slower network connections via low bandwidth channels multiplexed and synchronized into the source stream, or by streaming audio only. In the *Internet CNN Newsroom* project, audio streaming delivery could be provided to clients with ISDN bandwidth connections, where before no streaming would be possible. This enhancement alone would provide significant flexibility in deploying *Internet CNN Newsroom* on a large scale.

Another significant motivation for the StreamObjects system is the ability to provide *advanced playback* functionality, described later in this document. Advanced playback gives the user the ability to skip to any point within the stream while viewing it. This gives a user capabilities similar to rewind and fast-forward, except that the user does not see the video as it scrolls by.

[5] The NMIS project does not attempt to be a production quality facility, but the research provides useful information for companies attempting to a provide a commercial grade system in the future.

Another motivation for creating the StreamObjects system is to provide more editing capabilities to the teachers that use *Internet CNN Newsroom.* Users can apply the StreamObjects system to take selected excerpts from several news stories and make a new story. Students could conceivably work with this new, more editable media stream. This parallels some of the *Digital Video Library* research done on the *Informedia* project at Carnegie Mellon University.

Another project which hopes to utilize the StreamObjects system would be the *Shakespeare Project*, directed by Professor Peter Donaldson, at the Massachusetts Institute of Technology. The project currently archives various versions of some of the works of William Shakespeare. The StreamObjects system could be used to allow users to choose certain phrases out of Shakespearean plays, choosing the video and/or audio that corresponds to the section desired by the user. To produce this kind of resolution, all that is required is a index table listing each critical section's time offset. It is conceivable that by using technologies developed at places such as MIT's Media Laboratory one could automatically generate such index tables. However, such projects are beyond the scope of the author's Masters research.

1.3 Digital Media: Storage and Delivery

The following section describes several of the topics that revolve around digital media and its storage and delivery. An analysis is given to some of the issues surrounding digital media, an analysis of the MPEG standard, and a look at some of the existing architectures that can be used for digital media delivery.

1.3.1 Issues

Digital media storage and delivery issues are at the core of the StreamObjects system. Digital media is defined herein to be all video or audio signals encoded in digital form and stored on digital media (such as hard disks, CD-ROM, etc.). Digital media, like all forms of data, can be transported over networks for dissemination or broadcast. Some of the issues that effect the quality, speed, and flexibility of digital media delivery are covered below.

1.3.1.1 Delivery

When delivering video or sound, it is clear to see that the amount of transmission capacity required to transmit video may vary with respect to time. For this reason, it is possible for an encoding scheme to be designed to have a varying bit rate to encode a given source. This encoding strategy however has the potential to increase the complexity of the encoder, and makes estimating network load induced by streaming digital media more difficult, since the bandwidth has the potential to vary with time. The types of media files handled by the StreamObjects system are assumed to be of a fixed bit rate, while extending the system to handle variable bit rate streams is well within the realm of possibility.

When video and audio are encoded together from a common source, it is important for the separate signals to be integrated and synchronized in some fashion. Without such integration the video and audio may drift with respect to each other, and assuring synchronization becomes difficult. On the other hand, because video and audio differ greatly in the style and format of the data, it is important to keep the video data separate from the audio data. Such a separation allows differing kinds and qualities of audio to be synchronized with several kinds and qualities of video. For this reason, a solution that provides easy integration, synchronization, and separation is desirable.

1.3.1.2 Compression

The nature of audio and video is such that efficient, compression strategies are possible. These compression strategies can take advantage of the nature of human perception of sound and moving images, and the redundancy of successive frames of video that is derived from motion correlation. An efficient scheme will capitalize on this. With such efficiency, the representation of data becomes more complex, making access into a random point within a video stream more complex. The same is true for audio.

Another significant issue revolves around the decoder complexity/bandwidth tradeoffs. The more complex an encoding strategy, the more bandwidth requirements are reduced. This however, increases the cost of the software and hardware required for proper, efficient decoding of the more complex signal. For this reason, an encoding strategy must be chosen that provides high quality at a reasonable encoding complexity, bearing in mind the increasing power of computers as time passes.

1.3.2 The MPEG Standard

Currently, the computing industry has several formats for transmitting digital media, among them Quicktime, Video for Windows, etc. These formats, while popular, are proprietary, and do not provide the proper performance tradeoffs described above. These formats do not work as aggressively to capitalize on motion correlation, and therefore require much higher data rates to achieve the same quality output. Another series of formats, developed by the Moving Picture Expert Group, or MPEG, has become accepted by the International Standard bodies ISO and IEC.

Since one of the main goals of the author is to develop technologies with non-proprietary standard technologies, MPEG-1, the format most commonly used to date for video and audio delivery, became the obvious choice for the StreamObjects project. MPEG[6] is also the format for the existing video libraries of many current research projects, including NMIS and its *Internet CNN Newsroom*.

[6] All further references to the MPEG standard throughout this document refer to MPEG-1.

1.3.2.1 Overview

The MPEG standard is defined within International Standards Organization documents ISO 11172-1 through ISO 11172-3. Each document defines the syntax, encoding and decoding strategies of each of the three kinds of MPEG streams: the video stream, the audio stream, and the system stream. The MPEG standard produces highly compressed video and audio streams utilizing several complex algorithms defined by each of the layers, resulting in extremely complex encoders, and relatively simpler decoders. This asymmetric strategy is fine for a *broadcast model* of distribution, where there will be many decoders, owned by content consumers, and only a few encoders, owned by the producers of media content.

1.3.2.2 The Video Stream

The MPEG video stream standard defines a coded representation of video for digital storage. This representation takes great advantage of motion compensation in encoding video to attain high compression ratios, often exceeding 100 to 1 over raw digitized video. This highly compact form requires both a complex decoder and a significantly more complex encoder. An MPEG video stream often defines pictures in the stream relative to other temporally near picture frames to achieve this high compression ratio. Greater detail into the format of the video stream will be given in Chapter 3.

1.3.2.3 The Audio Stream

The MPEG audio stream provides a coded representation of high quality audio for digital storage. This representation takes advantage of sophisticated psycho-acoustic models to attain high compression ratios, typically near 14 to 1. Like MPEG video, the audio encoding strategy is asymmetric. An MPEG audio stream time slices audio data and encodes each small section of time separately and independently from all other time slices. Greater detail into the format of the video stream will be given in Chapter 3.

1.3.2.4 The System Stream

The MPEG system stream serves to synchronize and integrate many video and audio streams into a single multiplexed stream, while providing continuous buffer management. An MPEG system stream is capable of multiplexing together 16 video and 32 audio streams. Encoders capable of producing system streams must be capable of interleaving all the component streams into a single stream while preventing any of the buffers to the component streams from overflowing, while assuring that the playback of the given streams will be synchronized. For random access into a system stream, time stamp information is encoded throughout the system stream facilitating random access by decoders. For these reasons, the system stream is of critical importance to the StreamObjects system's architecture, as we shall see in Chapter 3.

1.3.2.5 Editing Streams

One of the chief deciding factors as to whether the StreamObjects system is feasible hinges on whether one can easily and efficiently *edit* MPEG streams. Throughout this document, *editing* refers to the process of *soft-segmentation*, the generation of an MPEG video segment from an existing larger one. This is in contrast to *hard-segmentation*, which creates segments of video at the time of encoding. Editing herein does not refer to rearranging or reordering video or audio, which is significantly more complex and to which the MPEG format is poorly suited for[7]. If it were impossible to edit an MPEG stream efficiently for whatever reason, the StreamObjects system would become impractical at best, inducing severe loads on the servers on which it would run. The entire performance model for the system assumes that one can easily edit MPEG streams.

[7] Section D.8.1 of ISO 11172-2 states most editing is best done before encoding is begun.

But is this assumption true? The answer is not so clear-cut. At each level of the MPEG syntax there are some regularities than enable editing to a certain extent, but there exist several restrictions that make ISO strictness hard to adhere to. Buffering issues are of primary concern, but because encoders are pretty relaxed in their conformance to the standard, a strategy can be created which *should* work for the vast majority of MPEG files. This issue will be covered in greater detail in Chapter 3.

1.3.3 Existing Architectures

One of the major goals of the StreamObjects project is to create technologies that will work hand in hand with the Internet as it exists today, to the largest extent possible. Since the goal of the StreamObjects system is to provide dynamically scalable media over the Internet, it is natural to look at the World Wide Web, and related architectures such as the Common Gateway Interface which aided in the creating of the StreamObjects system.

1.3.3.1 The World Wide Web

The World Wide Web has become such a huge part of the hype surrounding the Internet that it is not uncommon for semi-knowledgeable people to confuse the Web with the Internet itself. The World Wide Web provides a uniform method of access to the millions of data items that exist on the Internet. Because of the power and flexibility of Web browsers, it is possible to use the protocols for transferring documents on the web, the Hypertext Transfer Protocol, to deliver digital media.

The Hypertext Transfer Protocol, or HTTP, is a stateless protocol. HTTP is stateless because each request can be handled in an isolated fashion. A Universal Resource Locator (URL) serves to identify each individual document or data item, and thus a consistent deterministic, stateless way exists for referring to each item of data. Most Web servers implement the HTTP protocol by mapping the path in the URL to a file system path located within the server, and the file corresponding to that path is transmitted. Thus, such a protocol is satisfactory for static information retrieval, but since the StreamObjects system is dynamic, some form of extension to this model must exist to handle dynamic media retrieval.

1.3.3.2 The Common Gateway Interface

The Common Gateway Interface, or CGI, gives the Web greater flexibility in serving data items and documents. By placing a program in the pipeline between an input file and request and the client, the middle script or program, or *gateway* provides the ability to generate dynamic files upon request. This model proves ideal for the StreamObjects system. With CGI, a user may incorporate query information into the URL, allowing the computer to dynamically generate information.

The Common Gateway Interface provides the perfect method for handling dynamic data retrieval. Because traditional Web servers such as the one used for *Internet CNN Newsroom* are already used to deliver media clips to clients, adding CGI support is trivial with the StreamObjects system. Virtually all standard Web servers, such as the one created by the National Center for Supercomputing Applications (NCSA) contain CGI support. The difficulty would simply to create binaries of the StreamObjects system for each platform. This problem turns out to be a simple one, as will be discussed later.

1.3.3.3 Media Delivery and the Web

Using the Common Gateway Interface over the World Wide Web is not the only methods for implementing media servers. Many proprietary protocols and strategies exist to provide media delivery. Many of these methods *require* proprietary servers and clients, which reduce the number of potential clients and restrict the extensibility of the server itself.

Proprietary protocols make portability a greater issue, since proprietary clients must be implemented for the entire range of client platforms. If special client software exist only for Windows or UNIX for example, Macintosh clients would be excluded. The StreamObjects solution avoids this concern entirely. Though some form of proprietary client program is required in the special case of advanced playback (in order to create appropriate StreamObjects queries), specialized client software remains optional.

Proprietary protocols however, do have uses, some of which are beyond the scope of this project. For example, the Common Gateway Interface is primarily a query-response protocol, and thus more interactive two-way issues such as dynamic bandwidth adjustment and frame-dropping become difficult or impossible to implement without proprietary protocols over agreed ports. However, the design of the StreamObjects system is modular enough to work with these protocols with minimal changes to the architecture.

It should be understood, however that since the Common Gateway Interface can satisfy most media database queries in a dynamic yet deterministic manner, CGI remains the central delivery interface for the StreamObjects system. Further analysis into the limitations of the CGI/WWW combination is provided later in this document.

1.4 The StreamObjects System

This section gives an introduction into the constraints, components, syntax, and potential applications of the StreamObjects system. Some of the topics covered in this section will be covered in greater detail in later chapters of this document.

1.4.1 Constraints

In designing the StreamObjects, there were several constraints and assumptions that were chosen to restrict the scope of the StreamObjects project. This does not imply that the author dismissed the other architectures or systems, and future research may explore some or all of the issues raised throughout this project.

1.4.1.1 Ease of Deployment

One key constraint made in designing the StreamObjects system concerned the deployment of the system into academia or the corporate environment. It was clear that the StreamObjects system should be easily installed across many platforms and environments, regardless of Web server, operating system, file system, etc. This was achieved by using the CGI interface on a "vanilla" WWW server. While more advanced or specialized servers could be used, restricting the interface to CGI ensured easy deployment.

Realizing that newer interfaces to the Internet are continually being improved and redefined, care was taken to abstract the delivery and file interface from the internals of the StreamObjects system. The StreamObjects system was designed in such a manner that implementing a new delivery and query interface would be quick.

1.4.1.2 Portability

A second key constraint considered during the preliminary design phase was to keep the system as portable as possible. Towards this end, the popular object-oriented language, C++, was chosen. This decision reflects the language's power, acceptance and support within the software development community, its suitability to the task, and the familiarity of the author with the language.

Throughout the initial design and implementation cycles, more and more advantages were taken by the author of the advanced features of C++, such as virtual functions. However, realizing C++ is a language that is still evolving, advanced and inconsistently supported and implemented features such as exception handling (throw, try, catch, etc.) and Runtime Type Identification

(RTTI) were avoided. It is possible to utilize these newer technologies to potentially simplify a few parts of the project, but doing so would be ill-advised until these new features are more globally accepted and implemented.

Toward this same end, only the standard C and C++ libraries and functions were used in implementing the project. Wherever possible, the *iostream* and related C++ libraries were used for string manipulation, stream handling, and input / output (I/O) concerns. Only standard POSIX function of the C runtime libraries were used in implementing the StreamObjects system. This simple coding style virtually ensured that the StreamObjects system works across several UNIX platforms, as well as 32-bit Windows (Windows NT and Windows 95)

As will be demonstrated in Chapter 2, user-level multithreading support becomes essential for the proper operation of the StreamObjects system. The choice of such a threading system specification was simple, yet finding proper implementations proved to be difficult and the hardest aspect of maintaining cross-platform compatibility. The StreamObjects system supports two major multithreading specifications: The Win32 multithreading specification and the POSIX multithreading specification (pthreads) which covers most multithreaded UNIX platforms, as well as Windows 95 and Windows NT.

1.4.2 Components

The StreamObjects system, in its final design, is composed of three primary subsystems: the ThreadObjects subsystem, the Packet / Agent subsystem, and the Adaptive Interface subsystem.

1.4.2.1 ThreadObjects subsystem

The ThreadObjects system provides simple primitive objects for threads. Since the ThreadObjects system needs to be implemented for *at least* two specific thread libraries: POSIX and Win32 (see Figure 2 below), reducing the number of threading primitives is a clear and obvious objective. In order to implement the StreamObjects system, only three specific object classes were necessary: threads, blocking flags, and mutexes.

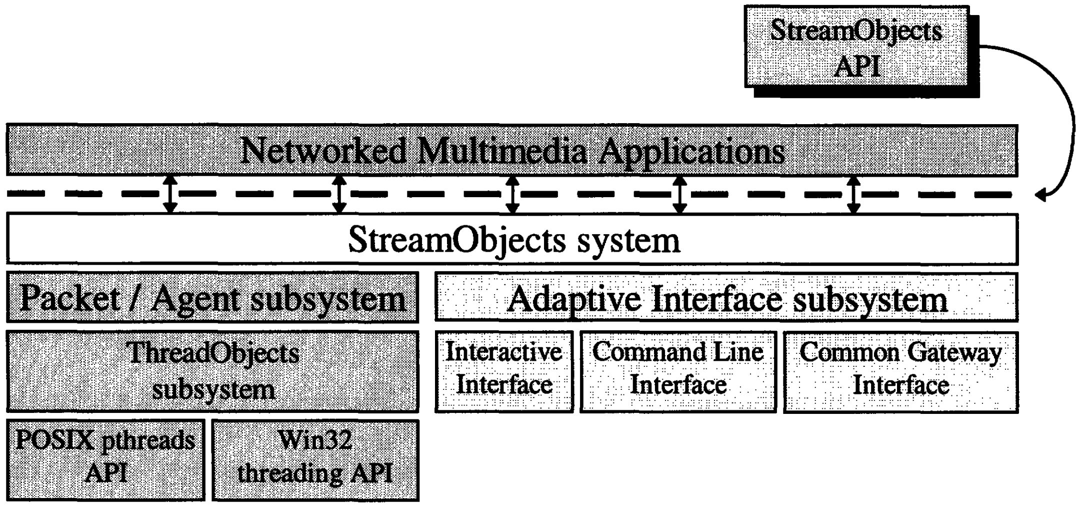


Figure 2: StreamObjects Architecture and API

The CThreadableObject class encapsulates the notion of agent-like objects running in their own separate threads. The CBlockingFlag class allows for individual threads to block, or wait, on a particular condition. This primitive is useful for flow control. The CMutex class describes a simple method for marking critical sections and controlling access to shared resources such as queues.

Together the components of the ThreadObjects subsystem provide all the tools necessary for orchestrating complex parallel processes. The ThreadObjects subsystem abstracts away the specifics of an operating system's thread implementation, and makes all other code modules that use the ThreadObjects system platform independent. A threading system, however pivotal, is but one part of the StreamObjects project.

1.4.2.2 Packet / Agent subsystem

The Packet / Agent subsystem provides a method for modeling parallel dataflow architectures in C++, building upon the multithreading primitives established with the

ThreadObjects subsystem (see Figure 2 above). A network of agents that pass packets (containing multimedia data, for example) to each other could be build to model the various behaviors of the StreamObjects system as a whole.

The architectures developed with this system can be applied to solving other problems. Packets can be subclassed to provide new kinds of data types that would enhance the functionality of the subsystem. The agent architecture is designed in such a manner so that rearranging agents to provide new functionality is both simple and straightforward.

1.4.2.3 Adaptive Interface subsystem

The adaptive interface subsystem, the third major component of the StreamObjects architecture allows a single StreamObjects executable to adapt to several different interfaces automatically. Currently, StreamObjects supports a command line interface, an interactive interface, and a CGI interface (see Figure 2 above).

The command line interface allows the user to specify the source and destination files as well as the query parameters through command line switches. This interface is consistent across the many UNIX and Win32 platforms for which the system runs. Some switches, such as the output filename, may be omitted, with the system thereby choosing appropriate default values.

The interactive interface allows the user to run the StreamObjects system without specifying any options — the StreamObjects system detects this and queries the user interactively, choosing the source and destination file, as well as the query used to process the file. The interactive interface also uses appropriate default values when certain settings are left blank.

The Common Gateway Interface, is perhaps the most useful interface, and the one the StreamObjects system was originally intended to support. When the StreamObjects system is called as a CGI binary file from any Web server, the StreamObjects system automatically detects this, and adapts to using the default output stream and input filename specified by the given

Universal Resource Locator. The StreamObjects system uses the query information of the URL, if given, to process the stream. The CGI interface cannot output to a specific file — only to the requesting client, to prevent the client from having the ability to write files on the server's file system, which could pose a security risk.

1.4.3 Query Syntax

The query syntax used in the StreamObjects system was chosen very carefully. The StreamObjects system is complex, and because of its power, major extensions to the system are to be expected. In order to handle this, the interface was designed to be both simple and extensible.

1.4.3.1 Simplicity

The interface to the StreamObjects system was based on a list of *key-value* pairs, perfect for the CGI interface which uses the same. This interface is also used in Windows setting files (.INI files), and since all the command parameters are human readable, debugging and manually adjusting these parameters is straightforward. This CGI-style interface also allows defaults to be overridden from a calling web page or application, without cumbersome initialization files.

1.4.3.2 Extensibility

The StreamObjects system's query syntax can easily be extended and upgraded. To do so, new key-value pairs can be introduced, or old pairs can extend the syntax of the value to handle new cases. Such a design prevents complications when new behaviors are added to the StreamObjects system.

1.4.4 Other Applications

The objects and classes created for the StreamObjects system have applications beyond that of server-resident, dynamically segmented media delivery. The StreamObjects system required many subsystems which can work harmoniously in other projects — whether related to media streaming, or not. Below are some other potential applications of the StreamObjects system.

1.4.4.1 Client-side Stream Interfacing

The StreamObjects system can be used for client-side stream interfaces as well. As de facto standards such as Open MPEG-1 (OM-1) stabilize, streaming files from a standard Web server would become simple. Such a client could be written as a helper application (such as NMIS's Netplay) or as a Netscape (a popular Web browser) plug-in or an OCX plug-in (for Microsoft's Web browser, Internet Explorer). Since client-side streaming is also best modeled using a pipeline of agents, the application of the StreamObjects system is self-obvious.

Using the StreamObjects system on both the client and server sides provides a uniform API for extending the media pipeline used in streaming. Possible extensions to this pipeline include adding encryption and decryption modules on both the client and server side, as well as applications for billing and caching. Using the StreamObjects architectures for billing is of great interest to Carnegie Mellon University's *NetBill* project, and coordination with CMU is expected in the coming months.

In addition, the StreamObjects system will be used in the current enhancements being made to NMIS's Netplay by Jeffrey Burstein, an undergraduate researcher working on the NMIS project. This will greatly enhance the flexibility of the client streamer, as well as add the ability to provide advanced playback features where they were impossible before.

1.4.4.2 Network Modeling

The StreamObjects system can also be used for network modeling. At the NMIS project, research is often done into what constraints exist in the distribution of video, audio, and other high-bandwidth multimedia applications over varying network architectures. The Packet / Agent subsystem and the ThreadObjects subsystem could easily be adapted to model these networks. This could be used to create accurate simulations of network load induced by various demand situations.

1.4.4.3 Computing Architecture Simulation

Another potential application of the technologies developed in creating the StreamObjects system would be for testing, debugging, comparing, and simulating computing architectures. Comparisons could be made between parallel dataflow architectures and traditional Von Neumann computing machines. The architectures developed here would also be useful for simulating control structure systems and other real-time systems.

1.5 CONCLUSIONS

Throughout this chapter we have been introduced to the background and motivations for the StreamObjects system, and what such a system can provide for server side scalable media delivery. A hint towards future usage of the StreamObjects system has been discussed, with more to follow in later chapters. We now examine the architectural evolution of the StreamObjects system.

Chapter 2: Architectural Evolution

2.1 Introduction

The StreamObjects system described in the first chapter did not simply "become." The StreamObjects system underwent several design-test cycles before reaching a stable architecture. Throughout the many design-test cycles, there was an overall trend towards more generality and increased power and flexibility. Since the final architecture can be used for many purposes beyond the intended purpose of the system, this chapter describes the architecturally relevant changes. Details of the MPEG-specific analysis will be discussed in Chapter 3.

2.2 Buffer Chain Model

In the earliest stages of design, it was unknown whether the MPEG data stream could be processed efficiently and simply in a single thread. Since single threaded code is the easiest to implement and port, such a solution was tried first. It was observed by the author that an MPEG stream could be modeled as a long, doubly-linked chain of memory buffers, each containing the given MPEG source file, in order, in its entirety. The entire stream need not reside in memory — the only buffers in primary storage (RAM) would be the section of the file that is being processed. This model was known as the *buffer chain model*, shown below in Figure 3.

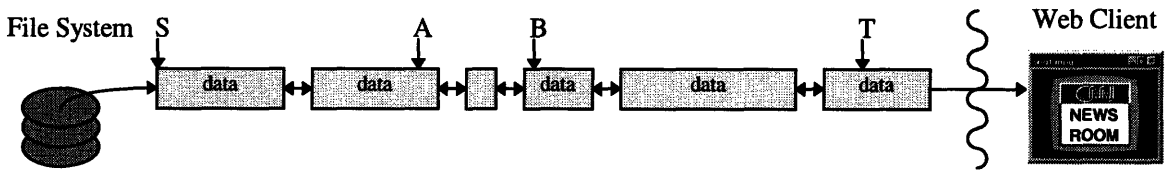


Figure 3: Buffer Chain Model

Editing the MPEG stream resulted in moving processing pointers from the beginning of the file to the end. These pointers are labeled S, A, B, and T in the diagram above, with S representing the source, T representing the sink, and A and B representing intermediary filters[8]. When a section of the MPEG file should be absent in the resultant file, that buffer area was skipped. When data should be inserted at the processing point within the MPEG file, a new buffer would be created and inserted into the buffer chain. A series of primitives were defined for deleting bytes, inserting bytes, passing along bytes, and peeking bytes. These primitives are similar to those found in any stream or buffer manipulation library.

2.2.1 Efficiency and Complexity

The primitives defined by this architecture are efficient in dealing with MPEG data. They avoid unnecessary copying of buffers if used properly. It is always possible to efficiently (i.e. in constant time) remove a section of the stream, even if the endpoints that denote the removal region do not align with a buffer boundary.

However, once it became clear that MPEG editing is too complex to describe with a single processing pointer, it became clear that the multiple processing pointers would have to maintain their own state, and would have to be synchronized by a central controller. It became difficult to think of a way of describing such a controller efficiently.

Since each of these processing pointers performed distinct tasks and needed to track their own state, the complexity of any controller became enormous. Attempting to control each of these individual processing pointers with a single thread would be inordinately complex. An issue arose regarding when pointers would overlap, and tracking which one was "in front" of the other, would be difficult and hard to debug.

[8] The convention of labeling source as *S* and sink as *T* and labeling all others *A*, *B*, *C*, etc. is used throughout this document.

2.3 DATA / AGENT CHAIN MODEL

To solve the problem of distinctly marking the place of each of the pointers, the notion of an agent object was created, for insertion into the pipeline. The architecture became a long chain of data and agents, with data being inserted in front of, removed behind or passed through the agents in the chain. A generic StreamObjects class was created, with special DataObjects and AgentObjects classes in between (see Figure 4 below).

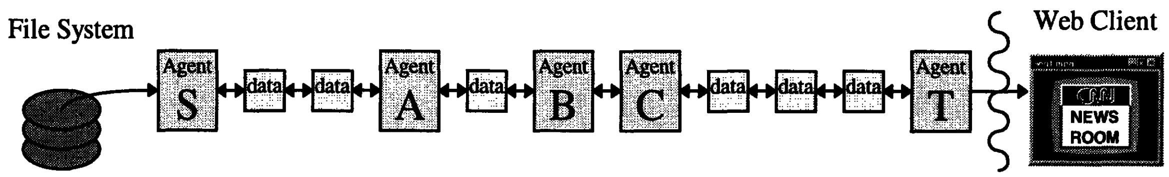


Figure 4: Data / Agent Chain Model

The ThreadObjects subsystem was developed at this stage of development. For the first time, the capacity to handle multithreading was necessary, and the ability to encapsulate threading into an agent object was desired.

2.3.1 Efficiency and Complexity

This model was the first model to introduce an agent architecture. The agent architecture proved to be an efficient way to partition the dynamic segmentation of the project into specialized individual agents. The classifying of the data into discrete objects of various classes was introduced at this stage, which encapsulated the behavior of data in an efficient manner.

The difficulty encountered with the data / agent chain model was predominantly related to synchronization issues. Two adjacent agents would have a doubly-linked list of data packets in between them. The first agent, or source, would be inserting data into the chain in front of itself while the second agent, or sink, would be removing data from the chain behind it.

To coordinate synchronization required some form of encapsulating all the data in between the agents. In addition, each agent needed individual pointers to pointing to the agent before it and in front of it, in addition to the inherited pointers to the object before it or after it, which could be a data packet or an agent.

It was quickly concluded by the author that data and agents need not share a common parent class — in fact, data should be completely separated from the agents that act on it. All agents could be encapsulated into a queue structure that would exist in between agents for transmitting data *packets*. This prevented exposing data in transit between agents. This resulted in the next step in the architectural evolution, the pipeline / packet-queue model.

2.4 Pipeline / Packet-Queue Model

The architectural evolution began to stabilize with the introduction of the pipeline / packet-queue model. Instead of viewing the system as a chain of data and agents, the architecture evolved to a pipeline of agents, with queues in between for passing packets (see Figure 5 below).

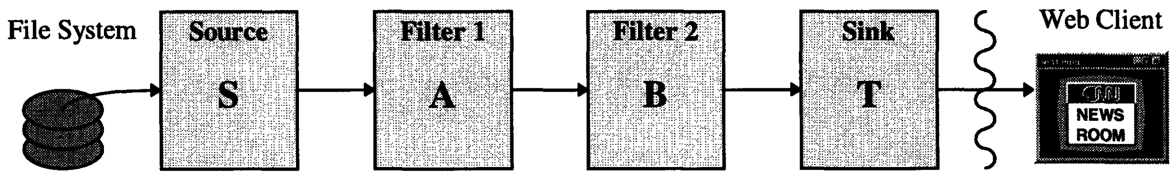


Figure 5: Simple Media Pipeline Design

Two distinct virtual parent classes evolved — the *CAgent* class and the *CPacket* class. Packet subclasses could be derived that would be tailored toward solving the given problem, such as media stream delivery. In the StreamObjects system, the key packet subclasses were the *CToken* class and the *CMemPacket* class. The *CToken* class provided a way of passing special symbols, tokens, or scalar values between the agents of the pipeline. The *CMemPacket* class provided a convenient method of passing chunks of media data between the agents of the pipeline.

With this architecture, agents subclasses could be derived for handling specialized, simple, well-defined tasks that when connected together in the proper order achieve a more complex goal. Common subclasses such as sources and sinks for various data types could easily be defined. All agents would have functions for receiving packets from their input queue and for sending packets to their output queue.

2.4.1 Efficiency and Complexity

The pipeline / packet-queue architecture provided for the most efficient and simple way of representing data while keeping it separate from the method of transport between agents to date. With the pipeline / packet-queue model the distinction between data and agents became enforced by making the two classes distinct, with no common parent class.

In addition, the introduction of the queue protected data in transit from being accessed by traversal from an incorrectly implemented agent. Packets in a sense are "locked" within the packet-queue, only accessible by pushing or popping packets from the queue. This architectural change encapsulates transmission control efficiently between the agents and queues.

Synchronization issues were resolved with the introduction of the packet-queue. Previous attempts at efficiently synchronizing data between individual agents were simplified because a single queues contained the synchronization primitives, no longer the agents surrounding the data. With the control mutexes and blocking flags residing within the queue, individual agents no longer needed to explicitly coordinate flow control — proper synchronization was implied by the access methods of the queue.

Queues were defined as an abstract class — they stored objects of the *CPacket* class, in an unspecified underlying structure. A particularly useful implementation of the *CQueue* class interface was the *CArrayQueue*, using a fixed-sized array to handle the storage of the packets. The *CArrayQueue* class became the default implementation of the *CQueue* class used in the StreamObjects system's architecture.

2.4.2 Flow control

With the queue providing a simple architecture for transmitting packets between agents, flow control between agents was still left unresolved. The StreamObjects system could conceivably use lots of memory resources in transporting MPEG data between the various agents. While queues provided a way of synchronizing data-flow between the agents, a method is still required for limiting the amount of resources used between agents.

In the StreamObjects system, the source is connected to the server's file system, while the sink is connected to the network. Since the network bandwidth is almost always the performance constraint and not the file system bandwidth, sink-based flow control is desirable. The amount of packets used in this model is only constrained by the size of the queues between the agents. If a linked-list queue is used, then the queue size is unconstrained — resulting in an unbounded number of packets in the system at any given instant.

To shift flow control from the source to the sink, a coupon-based ring architecture was design in which the sink is connected back around to the source (see Figure 6 below). The source would only be allowed to send packets along the pipeline when it receives a coupon from the sink. The sink would begin the cycle by introducing a fixed number of coupons into the system.

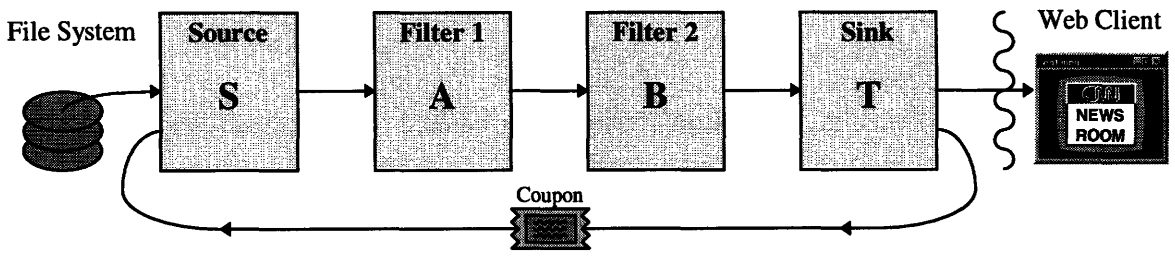


Figure 6: Coupon-Based Flow Control with Token-Ring

This method of flow control works well when the pipeline agents conserve the amount of packets flowing from the source to the sink. However, analysis of MPEG segmentation clearly shows that there no such conservation of resources exists. Agents at any place within the pipeline may choose to remove potentially large sections of data which contradict any notion of conservation. For this reason, a more global method of controlling the number of resources becomes necessary, as described in a later section this chapter.

2.5 Packet Agents: Directed Graph Network

One of the last major changes to the StreamObjects system arose from a desire to model the StreamObjects system more simply with greater granularity. The pipeline architecture assumes that all agents acts as filters — passing packets from input to output, occasionally removing, inserting or modifying packets along the way.

This structure, however, is unsuitable for describing agents that sort or merge packets. Such agents are inherently multiported, and would require multiple queues between agents — one for each connection between agents. Such an architecture is desirable for the StreamObjects system since the source file is usually multiplexed, and it is easy to conceive of an architecture where it would be important to handle, sort out, and process each stream separately.

In this new architecture, two new classes were defined to handle communication: the *CInport* class and the *COutport* class. Each *CAgent* object would have methods to allow access to each CInport or COutport. The CInport class had methods for connecting to the COutport class, and vice-versa. When an CInport object is connected to a COutport object, a CQueue is shared. Both port classes are responsible for managing the shared queue. The COutport class has a method for sending packets, and the CInport class has methods for receiving packets. These functions enforce directed usage of the queues — it becomes impossible for a receiver to send or vice-versa, unless a new pair of ports in the reverse direction are created.

The change of the StreamObjects system architecture from a linear pipeline to a directed graph (see Figure 7 below) was a drastic and complex one. The class of problems that could be handled increased greatly, and many changes in the complexity and efficiency of the system occurred.

2.5.1 Efficiency and Complexity

With the new architecture, transmission of packets is no longer handled *directly* by the CAgent class. Communication functions have been relegated to the CInport and COutport objects. This makes the definition of the agent more complex and more simple at the same time. The agent becomes more complex since it can potentially receive and send packets over multiple ports. At the same time, agents become more simple because the agent no longer is responsible for sending or receiving packets directly. These architectural changes allow for the creation of multiplexers (muxes) and demultiplexers (demuxes), which become critical to the efficient design of the StreamObjects system, as covered in the following chapter.

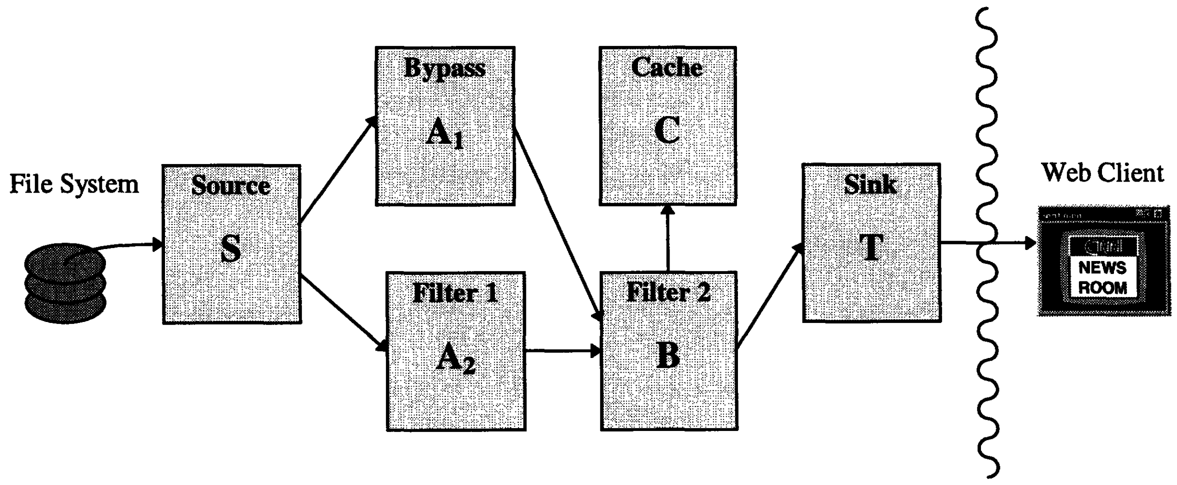


Figure 7: Packet / Agent Directed Graph Network

The pipeline of filters of the previous architecture was replaced with a directed graph of multiported agents. While this greatly enhanced the flexibility of the StreamObjects system, the vast majority of agents would still be linear, having only one CInport and COutport. For this reason, a simpler agent subclass was created, the functional agent.

This *CFnAgent* class implemented the protocol developed in the CAgent class but simplified the process of sending and receiving packets. The send and receive functions, in addition to residing within the CInport and COutport classes, were copied to the CFnAgent class. In the current StreamObjects system all agents but the multiplexer and demultiplexer were derived from the simpler CFnAgent subclass. Since the CFnAgent class inherits and uses the protocol of the CAgent abstract class, they can interact directly with any multiported agents with ease.

This final major architectural change provided the efficiency and simplicity of earlier architectures, while adding greater extensibility and flexibility needed to handle far more complex agent networks. Only this final architecture provides the ability to demultiplex an MPEG stream into its component streams, enabling each component stream to be filtered separately. Flow control nevertheless remains a major architectural concern.

2.5.1 Flow Control

In the final directed graph architecture, the number of packets in transit is only limited by the operating patterns of the agents that produce them, and the queues that connect them. Packets can always be sent as long as the queue is not full, and packets can always be received, presuming the queue is not empty. This enables the packets between all stages to be distributed optimally, without unnecessary constraints.

Nevertheless, the memory used by the StreamObjects system must be constrained, beyond the CArrayQueue size constraint. To achieve this, the notion of a global memory pool, which would distribute memory resources among agents, was introduced. Since the memory could be constrained globally, the StreamObjects system would operate in the most efficient way it can within the global memory constraint.

<u>2.6 RESOURCE CONTROL: PACKETS, POOLS, AND BLOCKS</u>

Global resource control is handled in the StreamObjects system by three interrelated objects: memory blocks, memory packets, and memory pools. With this architecture, the StreamObjects system can manage packets of data which can be dynamically split and reallocated (recycled). Memory is dispensed whenever needed; if not enough memory is available, the agent requesting memory is automatically blocked until its request can be granted. The following describes the individual component classes of the StreamObjects resource control system.

<u>2.6.1 Memory Blocks</u>

Memory blocks objects are defined by the *CMemBlock* class. Each CMemBlock object contains a character buffer of a size determined at initialization. When a CMemBlock is destroyed, its memory buffer is destroyed with it. A memory block differs from a traditional character buffers in the fact that it tracks the number of *memory packets* (defined below) that refer to it. Memory packets may refer to a section of memory, but the memory is actually allocated and deallocated by the memory block itself.

Whenever a memory packet wishes to use a memory block, the CMemBlock object automatically increases its usage counter. It is the job of a memory packet to call a member function of the memory block to indicate that it no longer is using it. This behavior proves to be critical in allowing several memory packets to share portions of the same memory block.

<u>2.6.2 Memory Packets</u>

The *CMemPacket* class is a derivative of the packet class, used for transporting chunks of media between agents. Though the major resource of the CMemPacket class is a character buffer, the StreamObjects system provides much more functionality than a simple buffer.

The memory packet internally handles memory block management. This is of critical importance when memory packets are split. In that case, two packets will share the same block. As described above, the memory block will have its usage counter increase with each new packet pointing to it, and when all the packets are recycled or deleted, the memory block is recycled or deleted as well.

The dynamic allocation and deallocation of blocks can be rather expensive operations since both involve heap manipulations. Many allocations and deallocations also result in potential fragmentation of the heap. To solve this problem, all allocations and deallocations for each memory pool are done at once.

2.6.3 Memory Pools

The *CMemPool* class is defined to distribute memory packets and blocks to all the agents of the StreamObjects system. The memory pool allocates a predetermined number of memory packets and another set of fixed-sized memory blocks. These packets and blocks are kept in two separate queues which act as dispensers.

The choice of a uniform block size is appropriate considering the media processing application at hand. The choice of block size does not prevent the existence of small packets. In addition to not utilizing the full size of the block, one can use the split feature to separate off a small packet from a much larger block containing two packets. This is in fact how the StreamObjects system utilizes the memory blocks, resulting in very low memory waste and overhead.

Despite the choice to use uniform block sizes for distributing memory, it is still possible to have variable block sizes by having multiple memory pools, each of a different size. Such flexibility is unnecessary, however, for efficient implementation of the StreamObjects system.

A key aspect of the CMemPool class is the "when available" distribution strategy. The same blocking CArrayQueue class used in agent to agent communication are used to dispense the memory blocks and packets. These queues automatically block when empty. Thus when the memory supply runs out due to an aggressive source (for example), the source will be blocked, and therefore forced to stop producing until an agent down the pipeline recycles the memory back into the pool. This strategy for resource control maximizes both performance and resource allocation, as described in Chapter 4.

2.7 Starvation and Deadlocks

Proper resource management must not only be efficient, but also be *correct*. In the current version of the StreamObjects system, it is possible to reduce the number of resources allocated such that the queues will empty such that no agent may continue. When this occurs, the StreamObjects system starves, and each of the agents are deadlocked. The occurrence of deadlocks in the StreamObjects system are fatal, and thus the system must be designed so that this becomes impossible.

Though reducing resources to intolerable levels result in deadlocks, a simple and straightforward design of the system can guarantee what level of resources should be required so that deadlocks cannot happen. In early tests, resources were set comfortably above this minimum resource level and deadlocks were avoided.

2.8 Conclusions

As this chapter concludes, it is important to reiterate the fact that the StreamObjects architecture was not arrived at instantly, but evolved through several architectural models that become increasingly more capable of handling the tasks of the StreamObjects system. Several of the improvements to the architecture of the system were influenced strongly by analysis into the format of MPEG system streams. The following chapter details this MPEG analysis, with special attention given to its effect on the ability to soft-segment MPEG streams in a efficient, low overhead manner.

CHAPTER 3: MPEG ANALYSIS

3.1 INTRODUCTION

The StreamObjects system provides dynamically segmented scalable multimedia over networks such as the Internet. As discussed in the first chapter, the StreamObjects system uses the digital video and audio standard agreed upon by the Motion Pictures Expert Group. This chapter analyzes the MPEG standard and its syntax, with emphasis given toward aspects crucial to editing such streams. Details regarding the representation of video or audio that are below the level necessary for editing are ignored for conciseness.

3.2 CONSTRAINTS

The StreamObjects system operates under several constraints. One constraints involve the level of detail in which an MPEG stream must be viewed, here referred to as *depth*. Another constraint requires that the amount of byte manipulation performed on the stream be kept to a minimum, in order to minimize server load.

The syntax of MPEG streams allow for great flexibility, making the job of decoders or editors very difficult. However, because existing encoders code MPEG streams with greater regularity than a *technically* legal stream that could be contrived, certain assumptions can be made about the size and frequency of packets by taking a few experimental measurement on the given MPEG stream before actually processing the stream.

Thus, the StreamObjects system processes MPEG streams in two distinct phases, the *preprocessing phase* and the *transmission phase*. The assumptions made about the MPEG stream do not necessarily restrict the format of MPEG streams. However, StreamObjects relies on the assumption that the encoder is a "reasonable" one, to ensure the efficiency of the system.

3.2.1 Preprocessing Phase

The first phase, the preprocessing phase, collects statistics and other relevant information about the MPEG stream. This collection of statistics is done by sampling the stream, paying close attention to the beginning and the end of the MPEG stream, allowing the preprocessing to be a constant time operation.

It is essential that preprocessing be a constant time operation with respect to the length of the source file. The amount of time spent preprocessing a stream adds directly to the latency before streaming starts. As mentioned in the first chapter, the StreamObjects system can be coordinated with a properly designed client to provide advanced playback functionality. Whenever a user wishes to fast-forward or rewind to a different place within the source file, the client must cancel the existing stream and open a new stream. If this latency between choosing a starting point and actual streaming of video to the client is too large, advanced playback becomes impractical.

Several steps were taken to assure that preprocessing can be done in constant time with the lowest latency possible. Firstly, preprocessing is only be done once for each request, by the source agent. Since each arrangement of agents includes a source, it is logical to place the preprocessing responsibilities with the source. The source agent is then dedicated to keeping and storing the results of the preprocessing, and making the results accessible to other agents via public methods.

Since MPEG streams generated by different encoders vary in the size, frequency and grouping of packets, it is important to make the preprocessing algorithm as general as possible so that variations between streams become irrelevant. The preprocessor was tested with MPEG files generated from several different encoders to satisfactorily demonstrate its compatibility and flexibility.

3.2.2 Transmission Phase

The second phase, the transmission phase, serves to process, strip, and adjust the source MPEG file into a dynamically generated MPEG output stream. For obvious reasons, the transmission phase must run in linear time with respect to the length of the input video segment. This is absolutely essential so that video files of any length can be processed by the StreamObjects system without inducing extra server load or delay.

In addition to requiring linear time operation, the transmission phase must process the video in a single-pass, with buffering as necessary. This single-pass operation may be achieved through many agents arranged as a several stage multiported pipeline. If multiple passes would be required through the stream, the latency would vary with the length of the file, which as mentioned above, should not be the case.

In addition to single-pass operation, the StreamObjects system has the advantage of buffering to simplify operation and relax the difficulty of implementing the system so that it runs in a single pass. The degree of this buffering should be minimized, however, for several reasons.

Increasing buffering has the potential to contribute directly to the latency of the system. As buffering increases, more of the stream can be fed into the system before being forced through the pipeline and directed toward the sink. This is because the source can process more information before overflowing the queues of the system, and the latency suffers accordingly.

Increasing buffering capacity increases memory requirements as well as increasing latency, and this therefore reduces the number of instances of the StreamObjects system that can effectively be run on a server. However, reducing buffering too much will cause the StreamObjects system to starve and deadlock.

3.3 Editing System Streams

The StreamObjects system is efficient because it performs the majority of manipulations at the system stream level. The system stream level serves to multiplex streams, and thus makes filtering particular channels out of the source stream relatively easy. The system stream also provides time codes called *system clock references* (SCR) with enough accuracy to be efficient for rough temporal segmentation. With the ability to simply perform temporal and channel-wise segmentation, understanding the system stream syntax is intricately linked to the design of the StreamObjects system.

The MPEG system stream syntax is defined in the International Standards Organization document ISO 11172-1. The system stream is composed of a sequence of *packs*, terminated by an *ISO-11172 end code*. Each pack consists of a one or more *packets*. Each packet contains data for a particular channel, except for the special *system header* packets, which contain information about the sub-streams that make up the system stream. The first packet of the first pack must contain such a system header (see Figure 8 below).

3.3.1 System Headers

System headers occur infrequently throughout a system stream. In the typical case, they occur only once, at the beginning, as required by ISO 11172-1. System headers contain information about the data rate of the system stream, the number and kind of component streams that make up the multiplexed stream, and other information useful for describing the format of the source stream. Unfortunately, not all the header information is present within the system header, so the StreamObjects must capture header information and statistics at various different *depths* of the source stream.

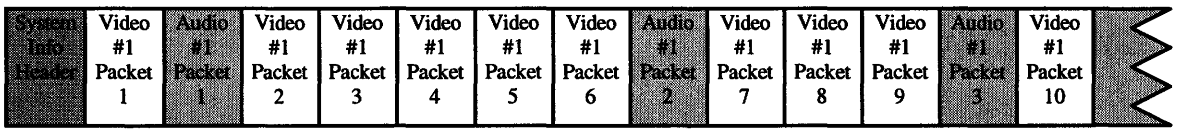


Figure 8: MPEG System Stream

The data rate (in bits per second) of the MPEG stream is present in both system headers and pack headers, and is referred to as the *mux-rate*. This rate may need to be adjusted to match the rate of the streams selected for output. To compute this rate requires information calculated from other parts of the stream. For this reason, buffering and multitasking are necessary in order for the StreamObjects system to be implemented in a coherent manner.

3.3.2 System Packets

All component streams, whether audio or video, are multiplexed together into a system stream. Each system packet, roughly two kilobytes in size, corresponds to a particular channel of data, or contains padding or system header information (described above). One of the purposes of the StreamObjects system is to filter unwanted channels out of a multiplexed stream, generating a new "narrower" MPEG system stream at a lower bit rate.

The removal of these channels is easy accomplished at the system layer. Each system packet is begun by an channel identifying pack header. Thus, the MPEG system stream may easily be sorted by individual component streams, and unwanted channels may be skipped and removed in their entirety, without costly analysis to determine the necessity of the contents of a particular packet. Since most packets require no complex modifications, removing unnecessary packets early help keep server load induced by the StreamObjects system as low as possible. The following sections describe the important aspects of the system stream syntax.

3.3.3 Pack Headers

Pack headers occur frequently throughout the data stream, and serve to both mark the data rate of the ensuing packets and the approximate display time of the enclosed packets. This data rate, the same as the *mux-rate* defined in the system header, is repeated in each pack header throughout the stream. This allows the MPEG syntax to handle the special case where the multiplexed stream changes bandwidth during delivery. Due to the added complexity of this situation, the StreamObjects system does not necessarily handle this special case.

The time codes included in pack headers are system clock references (SCR). The SCR time codes measure the passage of time throughout the system stream in terms of the *system clock frequency*, which is 90,000 Hz (for example, an SCR of 495,000 means 5.5 seconds).

Since MPEG system streams have fixed bit rates, the SCR corresponds linearly with the position with the multiplexed stream. For this reason, the SCR is approximately a time reference, since it does not refer to the time position within a *particular* video or audio stream. The precision is guaranteed to within 0.7 seconds to guarantee ISO 11172-1 compliance, which is generally suitable for random access within a stream.

To seek to a particular SCR time code, a Newton's method approximation algorithm is used. Since byte position within the stream is linear with regard to SCR, a particular SCR can be found usually within two or three iterations of the approximation algorithm.

Time codes need not start from zero for MPEG ISO-11172 compliance. For this reason, time codes are not linearly shifted when the StreamObjects system cuts from within the middle of an MPEG stream. This reduces the amount of byte manipulations necessary, keeping the StreamObjects system simple. The mux-rate may require modification for every pack header to comply with any change in bandwidth caused by stream selection. These are simple, one-pass modifications, linear in the length of the MPEG file — consistent with the transmission phase.

3.3.4 Packet Headers

One to several packets are found within each pack. Each packet is begun by a packet header. Since packet headers occur *at least* as frequently as pack headers, any packet header modifications must be simple, one-pass, modifications.

Packet headers are usually small (less than 16 bytes) and contain stream identification information, and sometimes channel specific time stamp information. These time stamps provide a more accurate indication of what time code corresponds to a particular packet of a particular channel (a particular display frame or an audio segment.) These presentation time stamps (PTS) and decoding time stamps (DTS) are not currently used in the StreamObjects system. The SCR time codes, while less accurate, are easier to seek with, and provide enough resolution for reasonable, easy operation of the StreamObjects software.

The stream identification code that begins each packet header makes it easy to sort packets by their corresponding stream. This enables each stream channel to be handled by its own agent. Potentially many agents could be chained to handle and process each individual channel independent of the behavior of other channels.

Packet headers can also contain *buffer size* information essential for proper buffering of both video and audio streams by a decoding unit. This information must be present in the first packet of any particular channel in order for an MPEG decoder to consider the stream valid. For this reason, agents that handle individual streams copy this buffer information from the first packet so that it can be spliced into what will eventually become the new first packet after temporal editing. Some streams repeat this buffer information with each packet header, but the presence of such buffer information cannot be guaranteed. Thus, copying buffer size information from the first packet header is essential for all component streams.

3.3.5 Packet Data

Packet data composes the vast majority of the bytes of the system stream layer. This is logical since all other bytes are overhead, which should be kept to a minimum while still providing the ability to have random access throughout the file.

The system layer only provides a method for multiplexing several heterogeneous channel types. The breakpoints between packets have no significance when viewed from within the syntax of each individual channel. Data is broken into packets of roughly uniform size (usually around two kilobytes in size). For this reason, important stream-specific information could be broken up across two or more packets. This means that stream-specific agents must either reconnect data across packet borders or otherwise take the fragmentation into account.

Any byte manipulations to packet data are stream-specific, and should be kept to the beginning point and the end point of temporal editing wherever possible. The two major stream types that are multiplexed by system streams are video streams and audio streams. The following sections describe the functionality of video and audio editing agents.

3.4 EDITING VIDEO STREAMS

The MPEG video stream syntax is defined within ISO 11172-2, the second volume of the MPEG standard specification. The video stream itself is a valid stream format supported by many MPEG hardware and software decoder solutions. The video stream itself contains its own temporal guides and marker codes which are independent of any headers in the system stream.

The video stream consists of a sequence of *picture groups*. Each picture group consists of several picture frames, usually 12 to 15, (one half a second) which can be decoded independently of any other picture group. There are three primary ways of encoding a frame, each with a different degree of compressibility and degree of dependence on other frames.

3.4.1 Sequence Headers

Sequence headers commence each sequence of picture groups. Sequence headers often occur before each picture group in some encoding strategies, but are only guaranteed to occur at the beginning of the stream. For this reason, any process which edits MPEG streams for editing must store the sequence header information when it first occurs. Due to the size of sequence headers (several bytes) and the size of packet data (approximately two kilobytes), sequence headers are guaranteed to be in the first packet of any video classified stream.

Sequence headers contain the information necessary for describing the bit rate, resolution and frame rate of the video stream. The StreamObjects system does not currently interpret or analyze this information, as it is never necessary to edit it. Nevertheless is it essential for this information to be copied before the first picture group at the beginning point of temporal editing. This usually alters the size of the packet that commences the editing point, since there is no way to guarantee that the first packet will remain the same size. Because any growth in packet size will not be larger than the size of the sequence header, which is only several bytes in size, this does not interfere with proper decoding of the StreamObjects system presuming the packets are manipulated to properly indicate the changes in size.

3.4.2 Picture Groups

Video streams are highly compressible because they capitalize on the fact that moving images are often temporally redundant and involve little or no change from frame to frame. For this reason, not all picture frames are encoded in their entirety — often frames are encoded based on preceding or ensuing frames. This technique is referred to as *motion compensation prediction.*

This strategy produces highly compressed video, but causes a chain of dependencies from one frame to the next that limit random access to any point within the stream. To solve this the MPEG video standard limits the length of the dependency chain by forcing non-interpolated frames at least once a second. These intra-coded pictures, or *I pictures*, must begin each picture group, and guarantee a minimum resolution (usually to within a half-second) for easy random access.

The StreamObjects system capitalizes on the random-access nature of I pictures to form easily targeted starting points for video streams. Picture groups, however, are not necessarily perfect random-access points. Picture groups may be *open* or *closed.* Open picture groups use information from the last frame of the prior picture group to decode the first one or two frames of the picture group. This improves the compressibility of the video stream, but causes a problem when editing. To solve this problem, the MPEG syntax allows the setting of a *broken link* bit within the sequence header which properly handles the missing frames caused by editing. Not all decoder boards fully support this bit, as covered in Chapters 4 and 5.

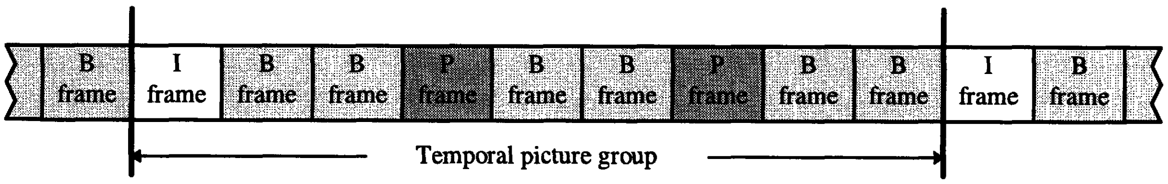


Figure 9: MPEG Video Temporal Frame Pattern

As mentioned above, there are three types of pictures: intra-coded pictures, predictive-coded pictures, and bidirectionally predictive-coded pictures. These three are referred to as I-frames, P-frames, or B-frames, respectively. In a picture group, the order in which pictures are displayed is almost always different from the order in which the pictures are accessed. This is because certain frame images are needed first in order to generate the other dependent frames. Thus, a decoder must use a reordering buffer to keep the display order — such a buffer is unnecessary in the StreamObjects system since the system does not tamper with the contents of individual picture groups. Nevertheless, a brief description of the frame encoding strategies is given below for completeness.

3.4.3 Intra-coded Pictures (I-frame)

Intra-coded pictures, or I-frames, contain the most information of all encoding strategies. An I-frame contains enough information to generate an image using information only from itself — I-frames do not have any temporal dependencies. For this reason, each picture group must contain at least one I-frame, occurring first, in access order. Picture groups could conceivably contain more than one I-frame, but in practice rarely do since they do not capitalize on motion compensated prediction and thus are the largest of the frame types in size.

3.4.4 Predictive-coded Pictures (P-frame)

Predictive-coded pictures, or P-frames, contain less information than I-frames. P-frames use motion compensated prediction from a prior I-frame or P-frame to achieve higher compression A typical picture group consists of several P-frames, each based on the prior P-frame, except for the first, which is based on the beginning I-frame.

3.4.5 Bidirectionally predictive-coded pictures (B-frame)

Bidirectionally predictive-coded pictures, or B-frames, contain less information than both I-frames and P-frames. B-frames utilize motion compensated prediction from the previous and/or the next I-frame or P-frame to generate their image. B-frames make up the majority of the frames of a typical picture group, and have the greatest temporal dependence and thus achieve the highest compression.

3.4.6 Complexity

Editing video streams in theory is easier than editing video streams in practice. Open picture groups are the most problematic. It is often that encoders do not properly detect and use the broken-link bit and attempt to decode the first B-frames which are based on the last P-frame of the prior (now removed) picture group. This usually results in the first 1/15 of a second of video to be partially garbled on some decoders[9].

3.5 Editing Audio Streams

The MPEG audio stream syntax is defined within ISO 11172-3, the third volume of the MPEG standard specification. The audio stream, like the video stream, is itself a valid stream format supported by many MPEG decoder hardware and software solutions. The audio stream syntax itself is extremely simple to interpret from a high-level decoder or editor. The audio stream consists of a sequence of presentation units, each begun by a syncword.

3.5.1 Presentation Units

In an audio stream, unlike a video stream, access and presentation (display) order are the same. There are *no* temporal dependencies between individual presentation units. Each PU can be decoded into a short duration audio clip. When decoded in succession, the presentation units assemble into an audio stream.

Presentation units need header information to describe the *format* and *layer* of the audio information they represent. Formats include monophonic, left channel, right channel and joint-represented stereo (two correlated stereo channels encoded together). The different *layers* of MPEG audio, represent different encoding strategies with different targets.

The most common layers tested with the StreamObjects system would be what the ISO specification calls layers I and II. The StreamObjects system will work, however with all formats and layers of MPEG data since this formatting information is unused and unmodified by the

[9] Attempts were made to modify the initial B-frames so that they no-longer referred to the previous P-frame (and could be then labeled as a closed picture-group). Such efforts were fruitless, and only serve to solve a problem that only occurs in some implementations, *not* a problem with the StreamObjects design or the MPEG video stream syntax specification.

system. Such header information exists at the beginning of all presentation units, so no header copying need exist from the first presentation unit in an audio stream.

3.5.2 Syncwords

All presentation units are roughly the same size, to within an error of one byte. The beginning of a presentation unit is marked by the presence of a *syncword*, a 12-bit pattern (0xFFF), always commencing at a word-alignment barrier. The typical syncword spacing interval, is roughly 500-700 bytes. Since packet sizes at the system layer are approximately three to four times as large, it is guaranteed that at least one syncword will be present in each packet, and thus each system packet will be marked by a PTS, or presentation time stamp. Such time stamps provide accurate timing and synchronization information to an MPEG decoder, but are unused by the StreamObjects system.

In addition, because a syncword is guaranteed to occur in each system packet, the detection of a safe editing point within an audio stream is quick, and guaranteed to be found within the first audio packet searched.

3.5.3 Complexity

Editing audio streams, as clearly shown above, is a relatively simple task. An editor's understanding of the MPEG audio stream syntax revolves primarily around the easy prediction and detection of syncwords within an audio stream. Since all presentation units are begun by regularly-spaced syncwords and each presentation unit can be decoded independently of any other, the task of editing MPEG audio is simpler than that of MPEG video. Decoders often capitalize on this by centering synchronizing decoders around the audio stream, and not the video stream.

3.6 The Editing Architecture

The evolution of the parallel dataflow architecture described in Chapter 2, was tailored for editing MPEG streams, though the architecture is suitable for several other applications, some with no relation to multimedia. The StreamObjects system is composed of several dedicated, multipurpose agents which when integrated form a powerful editing architecture.

The architecture can be adapted to accomplish several editing goals. Each request to the StreamObjects system includes a key-value pair identifying the task of the system. The three tasks currently supported by the StreamObjects system are *integration*, *info*, and *selection*.

3.6.1 Assumptions

The editing architecture makes a few assumptions about the nature of the input to keep the StreamObjects system running efficiently. The system assumes that the MPEG video and audio streams were produced by a "regular" encoder. By that, we mean that the packet sizes are relatively uniform and consistent. The system samples the first several packets to determine the maximum packet size; it is assumed that no unusually large packet will occur further into the MPEG stream and that the size of packets will not suddenly shift. Such assumptions are reasonable considering that MPEG video is encoded using high-speed, real-time hardware, and irregularly encoded video would introduce unnecessary complexity into the decoding and encoding process.

The MPEG syntax provides no way of guaranteeing bounds on the size of MPEG packets, but the assumption that the first several packets (at least five to ten) are representative of the entire data stream is a valid one for all tested encoders. It is also important to note that if a single MPEG stream produced by a certain encoder with certain parameters, all other MPEG streams produced by that encoder are guaranteed to work with this architecture. This again, capitalizes on the necessary regularity and predictability of MPEG encoders. Care was taken to test the

StreamObjects system against several encoders before finalizing the editing architecture and editing algorithms.

Though the StreamObjects system can handle MPEG files with many component streams (as many as ISO 11172-1 allows, which is 16 video and 32 audio), it is assumed that the number of streams present does not vary throughout the entire file. This assumption reduces the flexibility of the StreamObjects system, but MPEG files that violate this rule are extremely rare (none have been encountered by the author across various searches for test MPEG system streams).

3.6.2 Stream Integration

Stream integration, the first task intended to be solved by the StreamObjects system, involves synchronizing at least one selected stream (most commonly one video and one audio stream) together into a potentially lower bandwidth stream. In addition to such channel-wise selection, a temporal beginning point and end point may optionally be specified, the default being the beginning and end of the file.

This task takes advantage of the fact that the component streams were already to some extent synchronized and multiplexed — usually all that is required is manipulating the denoted bandwidth within the file, and paring away streams that have been not been selected for integration. This task is significantly less complex than integrating several streams that have never been synchronized. In that case, the size of system packets need to be determined, as well as the pattern of interleaving between the streams to ensure that video and audio buffers never starve.

Since the audio and video encoder has already synchronized the several streams, it becomes unnecessary to re-interleave the video and audio streams. Because of such complexities, the StreamObjects system does not currently support the ability to integrate two or more audio or video streams that come from two different source files.

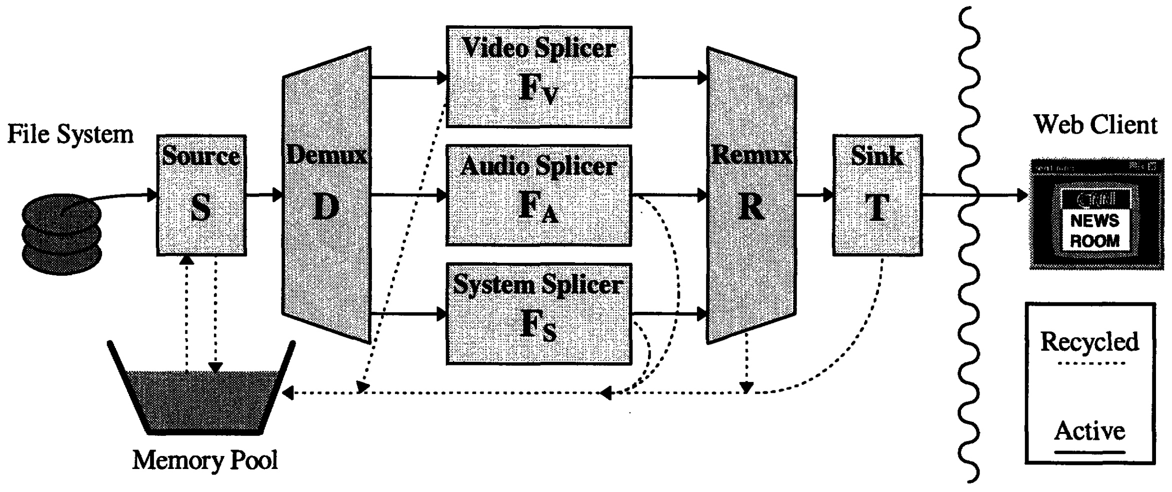


Figure 10: StreamObjects Integration Architecture

The StreamObjects system performs stream integration using a multiported pipeline (see Figure 10 above). The integration system consists of many agents. First is a source agent, which feeds the file from secondary storage to a demultiplexer agent, which separates each of the packets by their channel. Each channel has its own processing agent pipeline, which all feed into a remultiplexer agent which merges the component streams into a new output stream. This output stream is fed into a sink agent which routes the new stream to an output file or to a CGI-client.

3.6.2.1 Source Preprocessor

The first agent, the source agent, serves two functions. The first function is to preprocess the given input MPEG file. Since the source has direct access to the input stream, it has the ability to randomly access any point in the source file whereas other agents must accept data in sequential order.

The source preprocessor first parses the command syntax and stores it in a hash table mapping *keys* to *values* for quick and easy lookup. The hash table class used a simple modulus hashing function and handles collisions by double hashing as opposed to chaining. This is because the number of key-value pairs is usually small enough so that collisions are rare, and double or triple collisions virtually non-existent.

Once the key-value table has been constructed, the beginning of the MPEG file is examined, marking such things as the number of channels from the opening header, the spacing of packs and packets, as well as the packet to pack ratio. The number of packets per pack is used in turn to estimate number and size of blocks used for the memory pool, as well as the number of packets needed. The end of the MPEG file is also examined, so that the temporal range (beginning and ending SCR) of the MPEG file is accurately measured. All these statistics are accessible by public access functions of the source agent. Once the memory pool is initialized and all statistics are generated, the stream pointer returns to the beginning so the transmission phase of the source agent may commence.

3.6.2.2 Source Transmission

The transmission phase of the source serves to transmit the desired contents of the MPEG file in discrete manageable units. For this reason, the source transmits each system packet or pack header separately in its own memory packet. Since pack headers are small, they share a memory block with the first packet of each pack. The packet splitting function is used to separate the memory block along the border between the pack header and the first packet.

The source, through the preprocessing phase, learns which streams are to be kept and which streams are to be skipped. In addition, since the source agent is the only agent with random access capabilities, the source skips the file packets that are unnecessary, and only transmits essential packets.

To allow the source to only read the packets necessary, the source retrieves only the first few bytes of each pack header or system packet. The packet type is first identified and then the agent extracts the packet size, located within the first 18 bytes. The agent determines with that value the number of bytes to read or skip, and acts accordingly.

This editing algorithm not only efficiently transmits the packets, but labels each packet with a sequence number and the SCR from the nearest prior pack header. Since the output stream often begins at an SCR far from the beginning of the input file, it is important for the StreamObjects source to skip past the potentially vast number of packets before the starting SCR, described below.

3.6.2.3 Source Skip

The source skip feature of the transmission mode takes advantage of the random access capabilities of the source agent. Once the source transmits the first packet of each selected stream, the agent can then skip to the nearest SCR *before* the target starting SCR.

A Newton's method style approximation algorithm mapping byte interval to SCR interval is used to find the location of the packet nearest the target SCR. Such an algorithm usually converges on the appropriate SCR within one or two iterations. To prevent poorly formatted streams from keeping the algorithm from terminating, a limit on the number of iterations is imposed. Since most streams terminate well before the limit, this safety catch is only necessary for malformed or unusual MPEG files.

Once the target SCR is found, the source agents streams packets until the stopping SCR is found, which could be the final SCR recorded in the preprocessing phase.

3.6.2.4 Stream Demultiplexer

The stream demultiplexer agent routes system packets and pack headers to agents particularly suited to handling individual component streams. All streams are routed according to their channel ID, with all pack headers and non-component packets sent to *channel N*, the channel after the N component stream channels, numbered 0 to N-1.

The demultiplexer function is kept simple for several reasons. For one, it can be reused for other tasks, as shown in the next section. In addition, the simplicity and greater modularity kept the code simple and reduced efficiency analysis to the analysis of the performance of the packet queue subsystem.

3.6.2.5 Stream Remultiplexer

The stream remultiplexer agent serves to reassemble several component streams into a new, reintegrated stream. The reassembly of such packets would normally be problematic, since many channels must be merged in a defined order between several incoming channels, but the specific nature of stream integration defined above makes it simple.

It is important to note that the packets from the input stream were *already* multiplexed and in proper order. Since each packet is stamped and ordered uniquely, the remultiplexer can use a simple *merge sort* algorithm to create the newly ordered output stream.

One issue that arises in the remultiplexer involves the removal of packets such that two pack headers arise in sequence. This indicates that the packets between consecutive pack headers were completely removed, making the prior pack header unnecessary. To implement this properly the merge sort algorithm can be modified by adding a one memory packet delay when sending out buffers. The rule is simple — if the current memory packet about to be sent is a pack header and the new packet is also a pack header, do not send the old pack header.

It is possible to separate the functionality of removing consecutive pack headers into a separate agent attached to the remultiplexer, but such an architecture increases the number of agents without simplifying the code significantly. Introducing more agents has the potential of slowing down the StreamObjects system while requiring additional system resource.

In between the demultiplexer and remultiplexer, agents can be inserted to properly pare down individual component streams so that the resulting file is valid and usable by a wide range of MPEG decoders.

3.6.2.6 Video Stream Splicer

One such agent is the video stream splicer. This splicer serves to combine the first packet of a particular stream with the first packets after the starting SCR to produce a new valid stream commencing at the appropriate time reference. In order to accomplish this task, the splicer uses the following algorithm:

1. The video splicer keeps the first packet it receives.
2. It then waits for packets with system clock references higher than the starting SCR.
3. It determine if the output stream begins from the start of the input stream.
4. If so, it performs no modifications, and transmits the packets unaltered.
5. If not, the video splicer searches for the beginning of a *picture group*.
6. When found, the stream header information from the first packet is inserted before the start code, forming an appropriate start to the stream.
7. Any remaining buffered packets are recycled or transmitted.
8. The agent behaves like a straight through agent, passing all incoming packets on until the end of stream (EOS) is reached.

3.6.2.7 Audio Stream Splicer

The audio stream splicer performs a similar function to that of the video splicer. The syntax for MPEG audio is simpler to edit than video, so the audio stream splicer is in turn simpler as well. The algorithm for the audio stream splicer is as follows:

1. The audio splicer copies the buffer size information from the first system packet.

2. It then deletes all packets until the starting system clock reference is reached.
3. As with the video splicer, if the audio stream need not be edited from the beginning, the stream is passed through unaltered.
4. Once the target system clock reference is reached, all bytes before the first syncword of the packet are removed.[10]
5. Finally, the audio stream splicer passes on all incoming packets until an end of stream token is reached.

3.6.2.8 System Layer Splicer

The last splicer currently used in the StreamObjects architecture is the system layer splicer. The system splicer performs a simple task when compared to the tasks of the other agent splicers, yet the system splicer often must manipulate the majority of the packets it receives.

The system layer splicer is designed to connect to the miscellaneous *N channel* of the demultiplexer and remultiplexer. The only memory packets that may require modification are system headers and pack headers. System headers may require the alteration of the list of channels given, and both system headers and pack headers require the mux-rate modified to correspond to the new rate with streams removed.

Since system headers typically occur only once or twice throughout the stream, such modifications are simple and can be executed in constant time. Pack headers, however, occur throughout the stream and thus require modifications that, in total, require linear time. Nevertheless, the total computational load induced by the system layer splicer is relatively small.

[10] This is guaranteed to work since at least one syncword is guaranteed to occur within a system packet produced by a normal encoder.

3.6.2.9 Stream Sink

The last agent of the StreamObjects architecture is also the simplest. The stream sink serves as a gateway between the memory packet streaming architecture to the standard output streams used by the operating system.

The stream sink accepts all incoming data, writes its content to a file system stream, and then recycles the memory packet used to transport the data. This allows the StreamObjects system to be reused for several tasks, and can serve to integrate the StreamObjects system with other stream architectures, with input and output file system streams acting as gateways between the two systems.

3.6.3 Stream Selection

Stream selection is one of the most commonly performed and practical tasks of the StreamObjects system. The purpose of this task is to *select* a particular audio or video channel from an ISO 11172-1 compliant system stream and produce the appropriate video or audio stream, compliant with ISO 11172-2 or 11172-3, respectively.

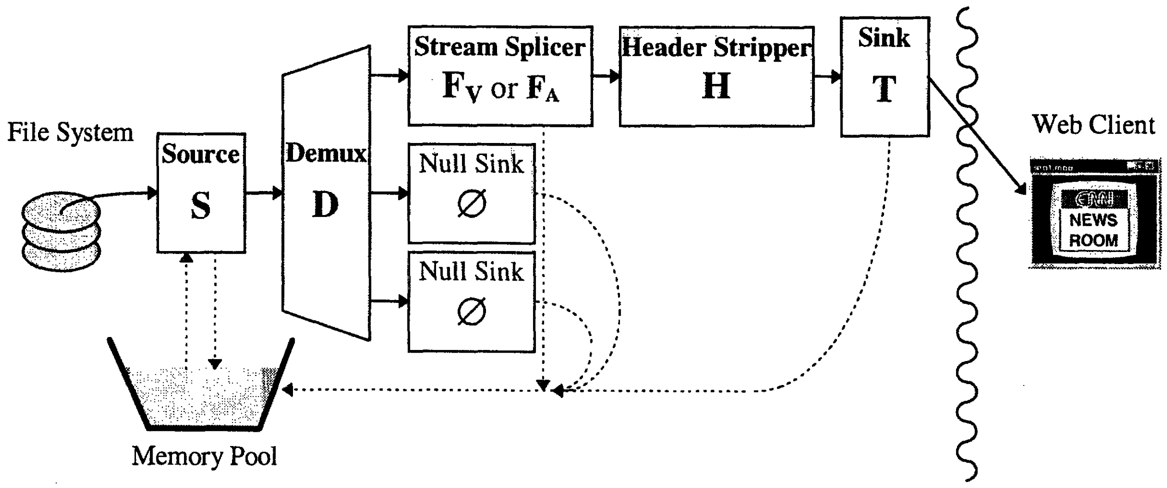


Figure 11: StreamObjects Selection Architecture

The arrangement of agents for the stream selection task is very similar to integration. The demultiplexer still serves to separate the component channels from the miscellaneous system packets, yet header stripper agents and null sink agents must be introduced to produce valid MPEG video and MPEG audio streams (see Figure 11 above).

3.6.3.1 Header Stripper

The header stripper agent is designed to be inserted after an audio or video splicer agent to extract the packet data from the header. This agent simply takes each system packet and modifies the memory packet so that is begins after the header. Such changes can occur easily with a constant number of operations per packet since the memory packet representation allows such changes by moving a character pointer and lowering a size member variable. Since all component streams use the same format for system packets, the same agent may be used interchangeably for both audio and video packets. The output of the header stripper may be connected directly to the same stream sink object used for stream integration.

3.6.3.2 Null Sink

The demultiplexer outputs a miscellaneous channel that is unnecessary for proper stream selection. For this reason, all packets routed to such a port must be absorbed and recycled. To accomplish this, a simple null sink agent was defined. All packets sent to a null sink are summarily deleted, until an end of stream token is reached.

This null sink can also be used to discard any data sent to unused component streams. In the selection operation, only one stream can be routed to the user at one time, thus all channels other than the primary channel (channel zero) are connected to a null sink.

3.6.4 Stream Information

The task of gathering stream information can simply be achieved by the source alone. Once a source is created, the preprocessor is automatically invoked, collecting valuable statistics useful for a client hoping to gain information about a particular stream.

Such stream information is formatted by the function that handles issuing the task. Once the information is properly formatted, it is routed as a plain text file (MIME type *text/plain* when using the CGI interface) to the client. Once the source is created, it can immediately be deleted since no transmission phase is necessary. This preserves the desired constant time operation of the stream information task.

3.6.5 Other Tasks and Architectures

The agents described above can be arranged to form different architectures that solve different problems. The stream selection task, for example, could be modified to have several header stripper / stream sink pairs so multiple streams could be routed to several files simultaneously. Such an architecture would be useful if one wished to design a program which demultiplexes a video and audio MPEG system file into its two component streams in parallel.

It is also possible to adapt the existing architectures to handle more specialized projects. Carnegie Mellon University's *NetBill* project could be incorporated into the StreamObjects architecture as an agent connected somewhere in the media pipeline.

In addition, specialized encryption agents could be introduced for producing secure, scrambled media delivery. Caching agents could be introduced to prevent recalculation of requests that occur frequently. Since a given URL always produces a deterministic MPEG output stream (assuming the input stream remains constant), it is possible to save common requests and bypass the above architectures altogether on common requests.

3.7 Conclusions

As shown above, the StreamObjects system provides a useful, simple tool for providing dynamically segmented scalable multimedia over networks such as the Internet. The above shows that it is clearly possible to modify an MPEG file with minimal modifications in the general case for usage across multiple MPEG standard decoders. The arguments for such efficiency of the StreamObjects system have been given, but real performance statistics need to be shown to fully prove the above arguments. In the following chapter we cover such efficiency arguments and performance analysis in greater detail.

CHAPTER 4: IMPLEMENTATION AND PERFORMANCE

4.1 INTRODUCTION

After building the system and assuring cross-platform capability, the question remained whether the system does indeed perform as expected. The system must be demonstrated to be practical by meeting the performance criteria proposed in earlier chapters.

This chapter details the implementation, testing and performance analysis completed by the author. Once the proper performance of the system is demonstrated, more questions about the future role of the StreamObjects system in media delivery can be asked.

4.2 IMPLEMENTATION AND TESTING

The implementation and testing of the StreamObjects system was performed over many design cycles which helped set the architectural course of the system. A development strategy was established early in the history of the project, with emphasis given to unit testing that would lead toward a common application programming interface. The StreamObjects API was developed in parallel with efforts to assure the compliance of the system to the MPEG standard regardless of the encoding method.

4.2.1 Development Strategy

In the creation of the StreamObjects system, a development strategy was chosen that capitalized upon the tools and platforms available to the author. Early on in the project it was noted that the system would be multithreaded, requiring a development platform that had such features.

Such a platform would also need to support the POSIX standard libraries and the *iostream* C++ libraries that would become part of the StreamObjects API. An integrated development environment, or IDE, would also be desirable since it would reduce development time and facilitate debugging especially for tight "test-and-change" cycles. The IDE chosen was *Visual C++ 4.0* for the Microsoft Windows 95/NT platform.

The Win32 API (common to both Windows 95 and Windows NT) supports multithreading and POSIX compliant standard libraries for C and C++, making porting to UNIX extremely straightforward. The Visual C++ and the Win32 API provided methods for discovering memory leaks and other hard to track bugs. With this ability, almost all debugging can be performed in Windows 95 and NT, with reasonable assurance that the code can be transparently ported across various UNIX platforms.

4.2.2 Unit Testing

Unit testing throughout the development phase assured the proper functionality of the StreamObjects system regardless of platform. The first units to be tested were elemental non-threaded objects. This discovered some of the *iostream* incompatibilities that exist between UNIX and Win32[11]. After such tests, a broad class of functions can be used across several platforms in the StreamObjects system without fear of incompatibility.

The next class of functions to be tested were the threading objects. These objects were first developed and tested for the Win32 platform. The number of primitives tested were small, to prevent unnecessary, exhaustive testing. Late in the development phase, these routines were ported to UNIX.

The emerging POSIX standard for threading on the UNIX platform extends the DCE (Distributed Computing Environment) threading libraries in existence for some time. The

[11] For example, the iostream manipulators *binary* and *text* exist only on the Win32 platforms, because such a distinction exists regarding carriage returns and line feeds. In UNIX, no such distinction exists, so *dummy manipulators* were created in the main include file to mask the differences between *binary* and *text* mode.

implementation the POSIX threading API chosen was developed by Christopher Provenzano at MIT. This implementation was chosen because it is both highly tested and freely available for several popular UNIX platforms[12].

The test functions that were being built into the StreamObjects source code proved extremely useful in debugging the POSIX equivalent. The ThreadObjects provided a uniform API to keep variances between Win32 and POSIX hidden. Implementation strategies such as these helped StreamObjects develop towards an important objective: a common StreamObjects API.

4.2.3 Towards a Common API

The standard POSIX libraries that cross between 32-bit Windows and UNIX form part of the StreamObjects API as it exists. The iostream libraries for C++ provide powerful functionality for manipulating character buffers and parsing data. Threading functionality was encapsulated through the ThreadObjects API, making threading issues transparent and platform independent.

Building upon the ThreadObjects and POSIX foundation, the StreamObjects classes for implementing parallel dataflow architectures simplify cross-platform development of enhancements and add-ons for the StreamObjects media pipeline. The future of the StreamObjects system will be covered in greater detail in Chapter 5.

With the common StreamObjects API firmly in place, it became increasingly unnecessary to test modifications across platforms. In the current development configuration, all changes and improvements to source code are made exclusively from the central, Windows 95 development station[13].

[12] MIT Pthreads implementations exist for SunOS 4.1.3, Solaris 2.3, DEC Ultrix 4.2, SGI IRIX 5.3, Linux 1.2, FreeBSD, NetBSD, and several other popular UNIX platforms.
[13] All changes to the source code are processed from the author's personal computer, running Windows 95.

Once the new system is fully tested from the Windows platform, the files may be transferred in their entirety with the click of a mouse to one of several UNIX servers where they can be compiled and ran, without error, without testing through a UNIX debugger. As it is clearly shown here, the StreamObjects API vastly enhances developer productivity.

4.2.4 Compliance Testing

Another important test of the proper functionality of the StreamObjects system involves the variations in encoding styles that exists across many different MPEG encoders. Packet interleaving patterns vary across several encoders, often requiring the source code to be adjusted to fully assure the compliance of the StreamObjects system regardless of encoding variations.

The current StreamObjects code functions properly across a diverse sample space of MPEG system streams, gathered from all corners of the Internet. Though some changes to the source code have occurred during the development phase of the project to assure full ISO 11172 compliance, the source code has essentially stabilized after detailed analyses into the formats of several MPEG encoders. With a stable source tree designed for extension and expansion, focus was shifted to assuring the performance criteria of the StreamObjects system.

4.3 Performance Analysis

The success of the StreamObjects system is inextricably linked to its performance in the tasks that it does. The StreamObjects system serves to process MPEG system files in real-time, requiring an efficient, powerful server that can provide the throughput necessary to handle several clients simultaneously, streaming data at the requisite bit rate.

For this reason, the many specific performance issues must be addressed. The preprocessing phase must be established as a constant time operation with regard to the length of the input file. Transmission time must be demonstrated to work in linear time with regard to the input as well. In addition, transmission throughput must always match or exceed the bit rate of the incoming stream. The effect of output bottlenecks on the performance of the system must also be analyzed. The choice of buffer sizes must be justified as well.

4.3.1 Benchmark Platform and Strategy

In order to address these issues, a benchmark platform was chosen to test the StreamObjects system. The StreamObjects system has been tested on various platforms, with each platform yielding throughputs that demonstrate the effectiveness and efficiency of the project in its current implementation. However, for official performance analysis, the SPARCStation 10 running SunOS 4.1.3 named *sunny.nmis.org* that serves as our test WWW server was chosen.

The choice of *Sunny* for testing was simple. This server/workstation was currently being used only to demonstrate the StreamObjects system, so the server had virtually no load or other factors that could interfere adversely with statistical analysis. Though a test web server is running on the NMIS system, the URL to the server is not published, and is thus primarily used for internal use and the occasional demonstration.

One of the major hypotheses of the StreamObjects system is that the primary inhibitor of performance of the system is not the system itself but the rate at which data can be retrieved from secondary storage (disk I/O bandwidth) and the rate at which data can be sent to the client through the network. For this reason, a performance testing strategy was chosen that involved transferring large MPEG system files to both the null device[14] and to local storage for output.

The files used for testing were placed in a large (over 150 MB) empty partition of a local fixed disk of the SPARCStation 10. When the output device is local storage, the output file is placed on the same partition as the input file. Variance in local storage is mostly irrelevant to the

[14] On the UNIX platform, the null device is referred to by the filename /dev/null. On the 32-bit Windows platform, also used for testing the filename NUL corresponds to the null device.

analysis of the StreamObjects system — the purpose of the fixed disk as an output device is to represent an external bottleneck that could slow the functioning of the system. Such a behavior was observed, as we shall see below.

To handle caching issues between test runs in a consistent manner, the same test case would be run several times in succession, timed using the UNIX time command which could provide timing estimates accurate to the hundredth of a second. Caching issues usually made the first timing somewhat high or somewhat low compared to all other timings. For this reason, the first timing of each case is dropped in calculating the average time. Any minimal effects that caching would cause on the performance are thus distributed evenly across all data points, and reflect common "real world" usage patterns for the StreamObjects system

A relatively large (404 MBit) standard test file is used for all tests (except for the size variance test, which uses files of various sizes) so that caches would be ineffective in significantly altering the timing results. Erratic timings were uncommon, with deviations for a single test limited to only a few hundredths of a second.

4.3.2 Stream Preprocessing Performance

The StreamObjects preprocessor, as described in Chapter 3, must quickly collect statistics about the nature and format of the given MPEG file. If such information was tedious to collect, or if the time to collect such information would vary with the length of the input file, alternative techniques would have to be introduced to achieve the same goal.

One such option would be to produce information files for each given MPEG system stream that would be utilized for each subsequent access to the same stream. While such an approach is simple and is easy implemented, the performance analysis makes such attempts useless. The StreamObjects system was given several MPEG files generated by the same *Optibase MPEGLab* encoder, the same files used for the *Internet CNN Newsroom* project. File sizes ranged from 24 MBits to 404 MBits, with each producing pre-processing latencies of less than 70 milliseconds (see Table 1 below). Such access time delays would easily be dwarfed by the delays introduced by the HTTP protocol over the Internet. No variance in pre-processing performance was detected among the various file sizes.

Informal tests with other MPEG encoders demonstrated similar pre-processing latencies, each producing latencies no greater that 70 milliseconds, regardless of file size. One difficulty of cross-encoder testing was the difficulty in finding MPEG *system* files produced by different encoders through the Internet and the World Wide Web. Nevertheless, all MPEG encoders tested produced strikingly similar results, validating the results gained with the Optibase MPEGLab encoder system.

File Size (MBits)	**Transmission Time (s)**	**Preprocessing Latency (s)**	**Throughput (MBits/s)**
24.58	1.86	0.07	13.22
61.87	4.56	0.07	13.57
74.88	5.83	0.07	12.84
133.08	10.51	0.07	12.66
175.25	13.96	0.07	12.55
283.84	22.71	0.07	12.50
404.15	29.40	0.07	13.75

Table 1: StreamObjects Throughput and Latency for Various File Sizes

4.3.3 Stream Transmission Performance

Another performance issue involved the linearity of transmission time with regard to the input length. As discussed above, the StreamObjects system must process MPEG files of any length at a relatively predictable rate that allows for streaming to network clients and local storage.

To test such performance issues, a selection of MPEG files of various sizes were streamed using default parameters (integrate both channels, for the entire length of the stream) to the null device. The results were tabulated (see Table 1 above) and then graphed. As the graph clearly shows, processing time to the null device is clearly linear in the length of the input file (see Figure

12 below). The SPARCStation had a null throughput of approximately 13.44 MBits/s. Such a high bandwidth clearly demonstrates that the limiting factor in the performance of the StreamObjects system on a typical modern server is the disk bandwidth.

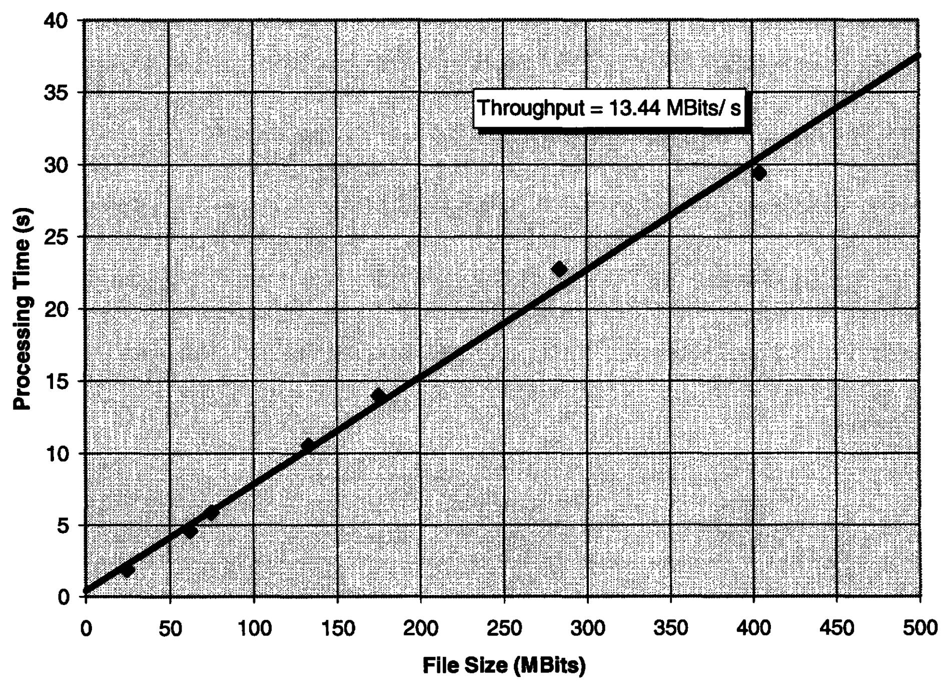


Figure 12: StreamObjects Processing Time vs. File Size

It is interesting to compare the performance of the StreamObjects system to other high-speed, low-load disk manipulation functions. To do so, the 404 MBit standard test file was copied to local store using both the UNIX *cp* command, the UNIX *cat* command, and the StreamObjects system. The StreamObjects system was configured so that by processing the file it would keep all present streams, thus producing an exact duplicate of the input file.

The three different test functions were timed both to local store and to the null device (see Table 2 below). The results were surprising — the StreamObjects system outperformed these standard commands (see Figure 13). The bandwidth measured when output was directed to local store was almost exactly half of the bandwidth when output was directed to the null device.

	Copying Time to Local Store (s)	Copying Time to Null Device (s)	Throughput to Local Store (MBits/s)	Throughput to Null Device (MBits/s)
UNIX *cp* command	82.46	0.10	4.90	∞[15]
UNIX *cat* command	78.11	0.10	5.17	∞
StreamObjects system[16]	59.72	29.40	6.77	13.75

Table 2: StreamObjects Performance Relative to UNIX File Commands

Such a performance would indicate that the system is almost always stalled by starvation external to the StreamObjects system. In the case of the null device, the secondary storage is used only for input, where when output is directed to the fixed disk, the storage device is used both for input and output. For this reason, it is expected that a performance degradation of *at least* 2 to 1 is to be expected. The fact that such performance degradation is almost exactly 2 to 1 demonstrates the efficiency (calculated to be 98.4%) of the system.

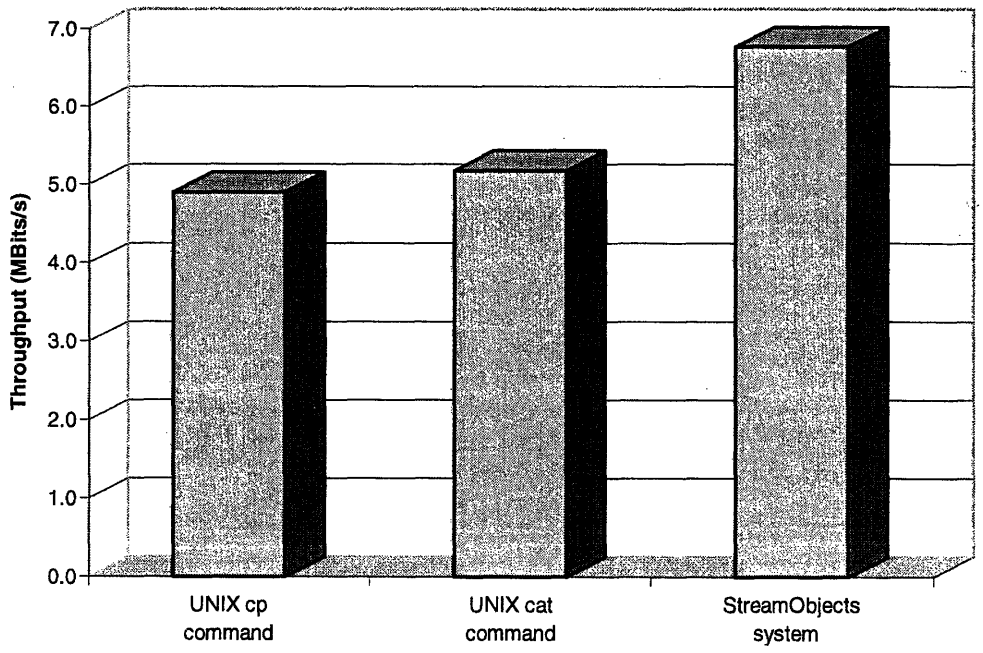


Figure 13: Throughput to Local Storage Performance Comparison

[15] The actual throughput is approximately 4000 MBits/s, which is assumed to be "infinite" since the delay is close to zero, irrespective of the length of the file.

[16] The StreamObjects system integrates the given file from beginning to end (the default parameters), which produces the same output as copying the file with commands like cat or cp.

The efficiency of the StreamObjects system is over 25% higher than the UNIX commands when measured on the SPARCStation. This performance difference could be accounted for by the efficiency of the standard libraries used by MIT's pthreads package, or by the sophisticated use of blocking and buffering within the StreamObjects system. Nevertheless, this performance test clearly demonstrates the suitability of the StreamObjects system to streaming applications.

4.3.4 Dynamic Segmentation Performance

The StreamObjects system does not always stream media files from the beginning. Dynamic segmentation performance ideally should be able to stream any part of any file with the same latency in a time proportional to the length of the segment desired. While this ideal is not completely attainable because of disk seeking issues, the StreamObjects system comes close to such a performance ideal.

The StreamObjects system was used to dynamically segment the standard test file in twenty different ways. The stream was segmented by channel in five different ways: by *integration* of audio and video, each separately or together, and by *selection* of video and audio. Temporally, the file was segmented four different ways, taking the entire file, either half of the file, and the second and third quarters (referred to as the *middle half*).

Temporal Range	Integration Time (s)			Selection Time (s)	
	Video & Audio	Video Only	Audio Only	Video Only	Audio Only
Entire File	29.40	23.89	10.80	22.87	10.51
First Half	14.74	11.92	5.53	11.36	5.29
Last Half	15.00	12.01	5.49	11.46	5.39
Middle Half	14.78	12.10	5.45	11.53	5.38

Table 3: StreamObjects Performance Timings to the Null Device By Task

The results of such tests confirm performance expectations. When an MPEG file is streamed to the null device, the transmission time is almost exactly proportional to the size of the temporal segment desired (see Table 3 above) .

Performance is not, however, proportional to the ratio of bandwidth desired to bandwidth present. This is inescapable because the input MPEG file must be traversed linearly (by skipping past undesired parts and reading to memory the sections desired) regardless of what percentage of the bandwidth is actually kept. In addition, when the StreamObjects system asks for a block of the file to be skipped, it may have actually been read into memory anyway because of the nature of block sizes in a modern file system. Ideal channel-segmenting performance would require a file system with no seek time, and of course such a file system does not exist.

Temporal Range	Integration Time (s)			Selection Time (s)	
	Video & Audio	Video Only	Audio Only	Video Only	Audio Only
Entire File	59.72	48.75	14.47	52.40	14.19
First Half	28.28	21.98	6.18	21.05	6.05
Last Half	26.34	21.13	6.34	20.07	6.08
Middle Half	27.00	21.26	6.18	19.88	6.14

Table 4: StreamObjects Performance Timings to Local Store By Task

The same battery of tests was performed by routing the output to local storage, producing similar performance results (see Table 4 above). As expected, the transmission time of each segment is directly proportional to the percentage of the file requested. These results were compiled into a table comparing throughput for each task (see Table 5 below).

Task	Output Size (MBits)	Stream Bit Rate (MBits/s)	Throughput to Null Device (MBits/s)	Throughput to Local Store (MBits/s)
Integrate Video & Audio	404.15	1.219	13.75	6.77
Integrate Video Only	339.52	1.024	14.21	6.96
Integrate Audio Only	64.65	0.195	5.99	4.47
Select Video Only	334.28	1.008	14.62	6.38
Select Audio Only	63.64	0.192	6.05	4.48

Table 5: StreamObjects Throughput By Task and Output Device

Analysis shows that throughput to the null device exceeds the bit rate of the output stream by at least a 10 to 1 margin. When routed to local store, the bandwidth is reduced roughly by a factor of two, as shown in the graph below (see Figure 14). The bandwidth required to read the stream from local store dominates the transmission performance of the system. This shows that the StreamObjects system can support roughly as many output streams as the file system can support.

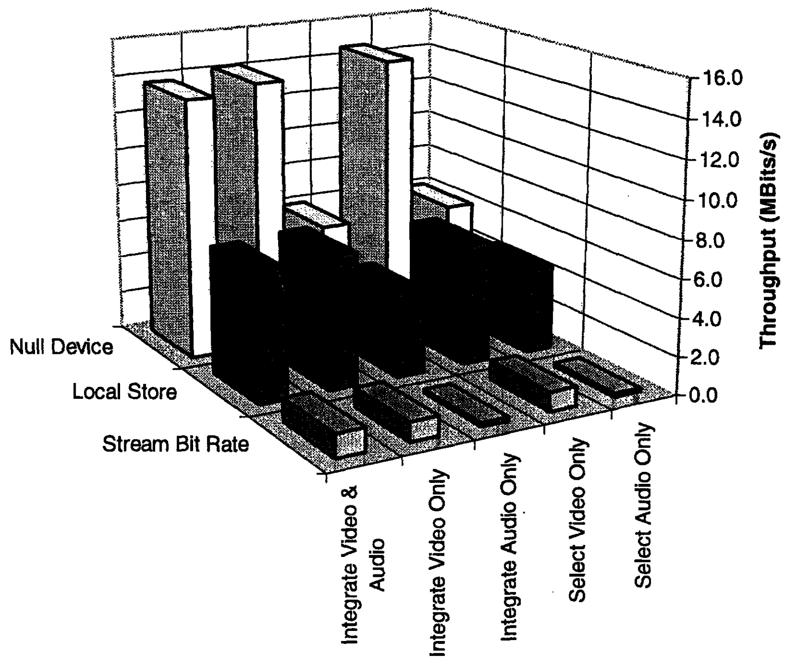


Figure 14: StreamObjects Performance By Task and Output Device

4.3.5 Platform Performance

Other performance issues revolve around the differences between platforms. Making such platform comparisons are beyond the scope and control of this project. The system has been designed in such a modular fashion that it should function on any standard C++ machine with a threading package. If a particular threading package is more efficient than another, it may affect the performance of the system, but other factors such as CPU processing power, I/O bus bandwidth, and operating system issues prevent the author from performing comparative tests across multiple platforms.

What has been confirmed is that the StreamObjects system can provide low latency preprocessing performance and high throughput transmission performance across all platforms to allow the streaming of MPEG data exceeding the 1.5 MBit/s second encoding rate at which ISO 11172 encoded streams are intended to operate.

4.4 Conclusions

In conclusion, we have satisfactorily demonstrated the performance of the StreamObjects system to provide dynamically-segmented scalable media delivery over the Internet. Whether within the confines of the CGI interface or without, the architecture provides a level of performance on all tested platforms that makes the StreamObjects API useful for high bandwidth media delivery.

Experimentation with memory packet issues and buffering demonstrated that the system could perform adequately even with as little as 64 kilobytes per stream. A content provider could choose a memory configuration appropriate for the bandwidth of the MPEG files it provides. Nevertheless, a content provider would rarely if ever have to increase the memory requirements of the system beyond their default settings.

The StreamObjects system was able to produce at least five 1.2 MBit/s streams simultaneous to clients, matching the performance of a server without the StreamObjects system. With the performance of the system assured, one can look toward the future of the project and its potential role in media delivery.

Chapter 5: Future Directions

5.1 Introduction

The StreamObjects project has succeeded in developing technologies for dynamic delivery of multimedia content in new and innovative ways. The software can be used to give content providers distribution options previously unavailable for this medium. With such potential, many other projects at MIT and beyond look to utilize the system and incorporate it in their delivery models.

5.2 Usage

The StreamObjects system can be deployed across many servers throughout the Internet community. The StreamObjects source code has been cross-compiled from the same source files for several UNIX and PC platform, and this will aid developers in mobilizing the project beyond NMIS. Any site on the Internet that currently distributes MPEG files through a Web server[17] can benefit by using the StreamObjects system to give their clients greater flexibility without increased storage costs.

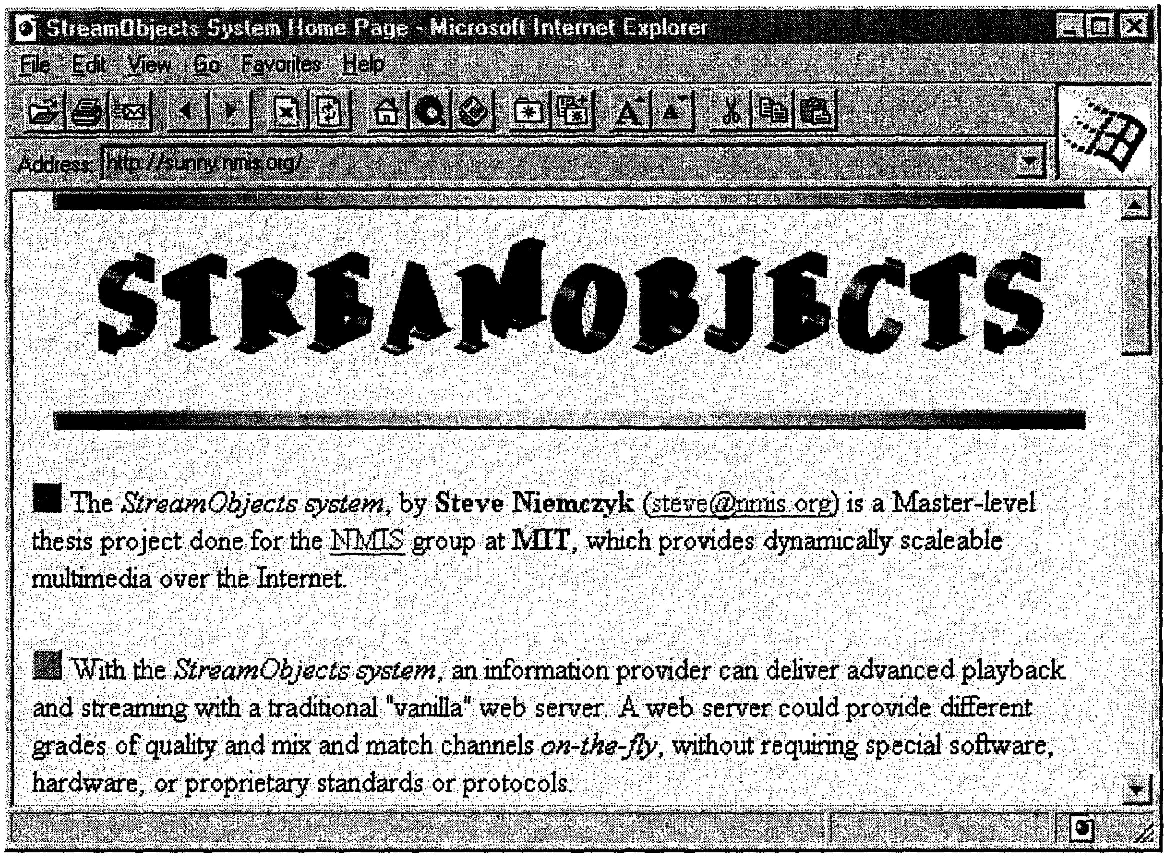
5.2.1 NMIS StreamObjects Demo Page

Currently, the StreamObjects system is used in a demonstrative setting within the Networked Multimedia Information Services project. The system is being used on *sunny.nmis.org*, a SPARCStation 10 running SunOS 4.1.3 running a "plain vanilla" Web server created by CERN.

A StreamObjects system home page has been created on the World Wide Web (see Figure 15 below) to both demonstrate the StreamObjects system and to serve as a centralized base for the

[17] Currently, versions of the StreamObjects system exist for virtually all flavors of UNIX, as well as Windows 95 and Windows NT.

latest information on the StreamObjects API and related technologies[18]. Binaries and source code will eventually be housed and maintained from the *Sunny* demo server as well as interactive pages which bring the innovative aspects of the system to light.


StreamObjects System Home Page - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address: http://sunny.nmis.org/
STREAMOBJECTS

■ The *StreamObjects system*, by **Steve Niemczyk** (steve@nmis.org) is a Master-level thesis project done for the NMIS group at MIT, which provides dynamically scaleable multimedia over the Internet.

■ With the *StreamObjects system*, an information provider can deliver advanced playback and streaming with a traditional "vanilla" web server. A web server could provide different grades of quality and mix and match channels *on-the-fly*, without requiring special software, hardware, or proprietary standards or protocols.

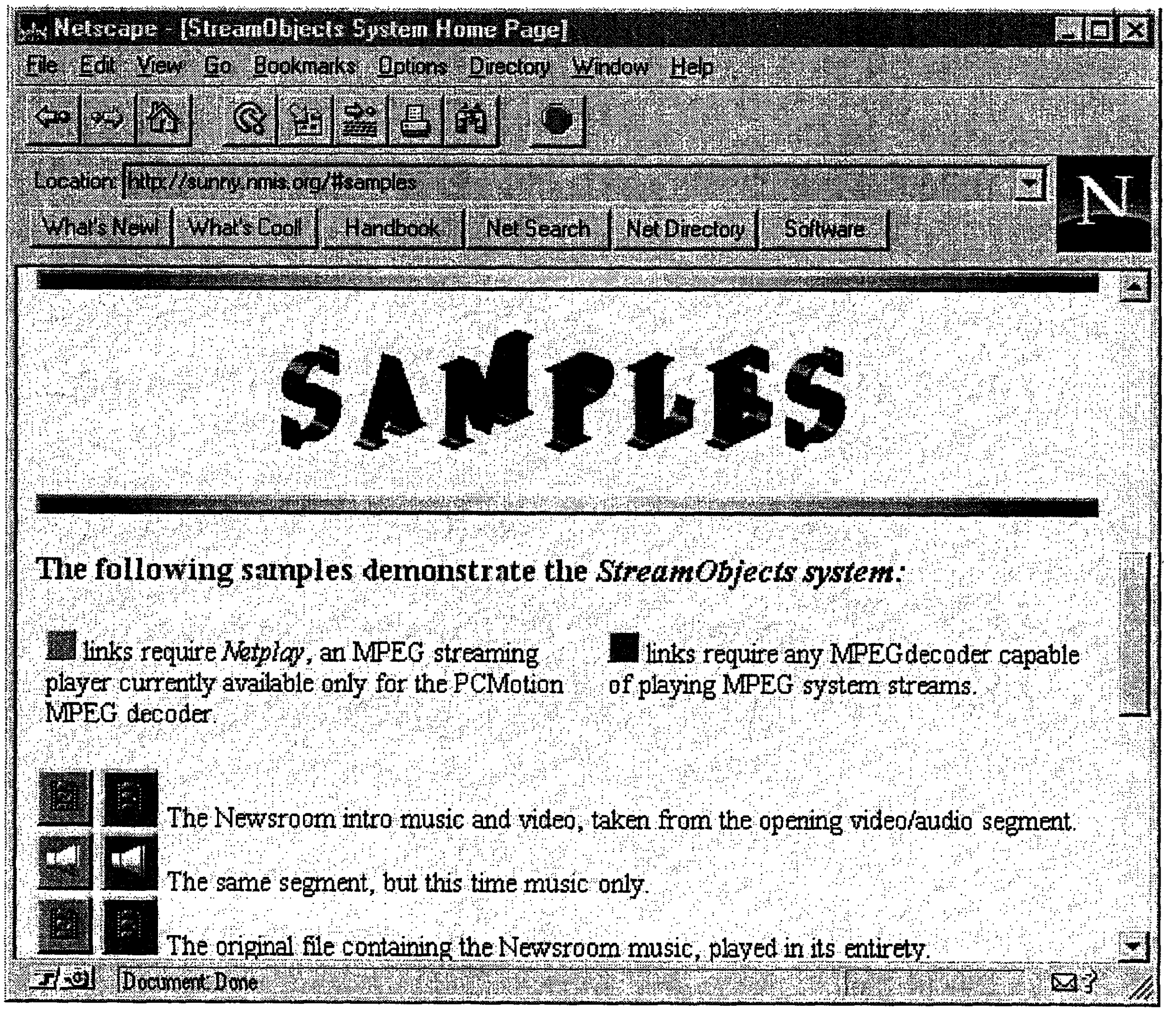
Figure 15: StreamObjects System Demo Page

Currently the StreamObjects demo page contains several sample links (see Figure 16 below) that demonstrate the dynamic segmentation capabilities of the StreamObjects system. The demo page provides support for more clients of all platforms than the original MPEG files it demonstrates could.

Some common MPEG decoder solutions, such as UC Berkeley's MPEGplay for many UNIX platforms and the Sparkle decoder for the Macintosh do not support system streams currently. For this reason, all MPEG files encoded by the *Internet CNN Newsroom* project would

[18] Eventually, when DCE threads are available for NMIS's main server, running AIX 3.2.5, the StreamObjects demo page will be moved to *www.nmis.org*.

be unusable for most Macintosh and UNIX users. However with a query as simple as "audio=none&task=select" the StreamObjects system can dynamically convert an MPEG system stream into a video only stream so that the Macintosh and UNIX base can currently be supported.


Netscape - [StreamObjects System Home Page]
File Edit View Go Bookmarks Options Directory Window Help
Location: http://sunny.nmis.org/#samples
What's New! What's Cool! Handbook Net Search Net Directory Software
SAMPLES
Document Done

The following samples demonstrate the *StreamObjects system:*

▨ links require *Netplay*, an MPEG streaming player currently available only for the PCMotion MPEG decoder.

■ links require any MPEG decoder capable of playing MPEG system streams.

The Newsroom intro music and video, taken from the opening video/audio segment.

The same segment, but this time music only.

The original file containing the Newsroom music, played in its entirety.

Figure 16: StreamObjects Sample Video and Audio Links

5.2.2 Internet CNN Newsroom

StreamObjects is currently being adapted for full deployment within the *Internet CNN Newsroom* project. The AIX 3.2.5 server being used by the project does not currently support threads, preventing direct support for the StreamObjects system, but plans have been set to add such POSIX multithreaded capabilities to the central server in the coming months.

Once incorporated directly onto the NMIS server, new delivery modes will be enabled for media distribution. All segments can be delivered as video streams, audio streams or synchronized system streams, *on demand*, without expensive storage redundancy. This allows any single MPEG system stream with a video and audio channel to provide three media formats instead of one. This not only allows a greater support for encoders that do not support system streams, but also provide support for streaming to computers with slower connections, such as 28.8 Kbit/s modems.

Currently a 5 minute video and audio clip encoded at 1.2 MBits/s would require over three and a half hours to download with a 28.8 Kbit/s modem. With the StreamObjects system the audio channel (encoded at 0.192 MBits/s) could be downloaded in just over a half an hour. With today's technology, such flexibility is in great demand for most content providers.

Future plans for *Internet CNN Newsroom* also include encoding the entire show as a single MPEG file, possibly in two passes each at different qualities. These files could be merged into a single four channel (two audio, two video) MPEG file that would allow a "mix-and-match" paradigm that would allow a user to choose the qualities and bit rate settings most suitable for her own computer system. Such a four channel file could provide two audio-only modes, two video-only modes, as well as four dual-channel delivery modes for a total of 8 useful delivery modes. With such a system, the number of delivery modes rise exponentially with the number of channels multiplexed together by the system stream. Another advantage of encoding an entire show as a single MPEG file, as discussed earlier, is that it eliminates the need to re-encode shows that were given poor segment time codes or shows that start too early or too late. In addition, links to short "teaser" video and audio files could be added to Web pages to give preview capabilities to users.

5.2.3 Client Side Development

Several applications of the StreamObjects system in the arena of client-side development are under consideration. The StreamObjects API provides technologies that can easily be adapted to client-side development, and because the architectures can be used across several platforms, streaming MPEG clients with advanced playback features can be developed for both 32-bit Windows as well as UNIX.

Currently the *Netplay* network streamer developed at NMIS will be adapted to use the StreamObjects architecture. The current *MediaObjects* architecture, developed by Jonathan Soo for the Netplay streamer, provides a simple pipeline model but does not support multithreading and is therefore limited in how it can be adapted for other projects.

5.2.4 Other Future Projects

One such project that wishes to be "plugged-in" to the StreamObjects system is the *NetBill* project at Carnegie Mellon University. The *NetBill* project introduces modules into the pipeline at both the server and client end that encode and decode the MPEG video and allow content providers to implement a "pay-per-view" streaming operation, similar to the delivery mode used over cable lines today. The multithreaded nature of the StreamObjects system makes dropping encryption and decryption agents into the pipeline trivial.

Another project with potential use for the StreamObjects system is the *Shakespeare Project* at MIT's Center for Educational Computing Initiative. By cataloging and indexing the entire works of Shakespeare, the StreamObjects system provides a perfect way to jump into the middle of a long video-file dynamically without breaking such a file into many segments. The author hopes to coordinate efforts with members of the *Shakespeare project* in the coming months.

5.3 Observations

In creating the StreamObjects system I have made several observations about the state-of-the-art in multimedia encoding, decoding, and editing. It has become clear that MPEG encoders and decoders need to adhere as strictly as possible to the MPEG specification, especially as changes are being made to make the specification more and more diverse and complex.

5.3.1 The Broken-Link Bit

One such issue that is poorly implemented by some decoders is the *broken-link* bit, described in Chapter 3 that allows for editors such as the StreamObjects system to break up an MPEG video stream at the border of an open picture group. With some decoders improperly ignoring such bits, video files encoded with open picture groups appear blotchy for the first two frames. Several decoders *do* however handle this properly, and greater steps must be taken in the future to ensure full compliance with the ISO 11172-2 specification as editing becomes more and more prevalent.

The issue also highlights some of the inconsistencies that can develop when a committee meets to design a portable standard. Original attempts at the MPEG specification wanted picture groups to remain entirely independent of each other. Such an encoding strategy would greatly simplify editing, and would preserve the notion of a picture group as an independent *presentation unit*. But pressures to improve the compression of video data came at the expense of easy editing and caused several MPEG decoding chip manufacturers to not support the broken-link bit (which arrived late in the specification). It is the author's hope that such miscommunications are avoided with future revisions of the MPEG specification.

5.3.2 Multithreading and UNIX

One issue that slowed the porting of the StreamObjects system was the difficulty of finding platforms that natively support user threads. The emerging multithreading standard, POSIX threads 1003 and beyond, is unsupported for many platforms, especially when being integrated with C++, or other relatively new languages.

Indeed, finding a platform that supported threads and C++ simultaneously proved to be a difficult task. Many current platforms have no publicly available implementation of POSIX threads, such as AIX 3.2.5. This made cross-platform development difficult, but the convergence of the standard and its appearance on several current platforms gives hope that future cross-platform software development will be smoother and less awkward.

5.3.3 Editing, Streaming, and the MPEG Specification

The ISO 11172 specification for the MPEG standard was designed to provide efficient encoding of digital video and audio, but was not designed with extensive editing in mind. While this is understandable, it is important to realize that the future success of MPEG as a standard will revolve around its ease to be edited and used across all platforms.

As of today, the MPEG system stream layer is only beginning to be gain popularity in the Internet community. As more companies such as *Microsoft* and *Netscape* provide greater support for streaming multimedia to the client, the demand for dynamic segmenting software such as the StreamObjects system will undoubtedly increase. Within just one year, it is expected by the author that several competing standards will exist for streaming MPEG content directly to a user's machine.

5.4 Conclusions

The StreamObjects solution to media delivery issues is a different one from many other solutions present in the Internet community[19]. The StreamObjects solution uses *standard, non-proprietary* techniques to achieve the goal of dynamically segmented scalable media delivery. As a non-proprietary solution, the StreamObjects system can permeate the Internet community just as the World Wide Web has these last several years.

[19] Such as XingTech's *StreamWorks* solution requiring specially designed servers *and* clients.

History has shown that proprietary solutions may provide excellent functionality, but only to a limited customer base. The trend in the Internet today towards open, multi-platform technologies such as Sun's *Java* clearly shows promise for the future of the StreamObjects system.

In the coming years many changes will come to the way we are delivered media and the way we access information content. As these changes occur, the StreamObjects system and its successors will pave the way for the changes to come and bring greater flexibility and stronger ties between multimedia and the ubiquitous Internet.

APPENDIX A: BIBLIOGRAPHY

Culler, David E. "The Explicit Token Store," *Journal of Parallel and Distributed Computing*, Cambridge, 1991.

CGI team. "The Common Gateway Interface," National Center for Supercomputing Applications, University of Illinois, Urbana - Champaign. Found on the Web at: http://hoohoo.ncsa.uiuc.edu/cgi/

Frost, Jim. "Portable Thread Synchronization Using C++," Software Tool and Die, 1995. Found on the Web at: http://world.std.com/~jimf/c++sync.html

McCool, Rob. "The Common Gateway Interface," Stanford University. Found on the Web at: http://icarus.stanford.edu:8050/cgi/overview.html

Motion Pictures Expert Group. *Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1.5 Mbit/s — Part 1: Systems*, International Standards Organization, Switzerland, 1994, ISO 11172-1.

Motion Pictures Expert Group. *Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1.5 Mbit/s — Part 2: Video*, International Standards Organization, Switzerland, 1994, ISO 11172-2.

Motion Pictures Expert Group. *Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media up to About 1.5 Mbit/s — Part 1: Audio*, International Standards Organization. Switzerland, 1994. ISO 11172-3.

Nygren, Erik L. and Whitson, Mike. "Introduction to UNIX Software Development," Massachusetts Institute of Technology. Found on the Web at: http://web.mit.edu/sipb-iap/unixsoftdev/www/

Personal Library Software. "Threads," Washington DC Area Smalltalk Users Group. Found on the Web at http://www.pls.com/dcstug/threads.html

Provenzano, Christopher. "Pthreads: General Information," MIT SIPB. Found on the web at: http://www.mit.edu/people/proven/pthreads.html

Tamirisa, Chary G. "Introduction to Multithreaded Programming," IBM Corporation. Found on the Web at: http://www.developer.ibm.com/sgi-bin/getobj.pl?/www/sdp/library/aixpert/nov94/aixpert_nov94_intrmult.html