QUALITY OF SERVICE ANALYSIS FOR DISTRIBUTED MULTIMEDIA SYSTEMS IN A LOCAL AREA NETWORKING ENVIRONMENT

A Dissertation Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Doctor of Philosophy

by

Edward Chi-Fai Chung

June, 1996

Acknowledgments

I wish to take this opportunity to express my sincere gratitude to those people who provided me with the guidance and encouragement during the final stages of this education endeavor. Special thanks is given to my dissertation advisor, Dr. Mehmet Celenk, for his patience, understanding and support throughout the project. His dedication to work and attention to detail never cease to amaze me. I am also grateful to my other committee members; Dr. Dennis Irwin, Dr. Jeffery Dill, Professor Hari Shankar and especially Dr. Costas Vassiliadis, for their time and valuable suggestions.

I owe much thanks to Mr. Sean Ann of RIM Communications Ltd., Hong Kong, for his invaluable support. Mr. Sean Ann is an exceptional friend whose suggestions are often inspirational and challenging. Partial funding of this research by RIM Communications Ltd. is also gratefully acknowledged.

Also deserving acknowledgment are the following individuals for their kind assistance: Dr. Jerrel Mitchell, Mr. Timothy Bambeck and Mrs. Janelle Baney. Mrs. Denise Ragan who gave her time to assist me with things that made my life much easier. Special thanks go to Ms. Lisa Lung for her spiritual and emotional support. She always reminded me of those things which are really essential in my life that I sometimes overlooked.

Lastly, my heartfelt gratitude goes to my parents: Dora and Eric Chung, for their encouragement, love , support and most important of all, friendship that I cherish now and will treasure forever.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Multimedia Networking Versus Traditional Data Networking

Distributed multimedia networking environments are an emerging application domain [1, 2, 3, 4, 5, 6, 7] that introduces new challenges to distributed processing systems (DPS). Such environments are characterized by the presence of groups of users, connected via a computer network, cooperating to achieve common tasks through sharing mixed-media information [8] such as text, graphics, facsimiles, data, audio, and video. While typical database systems handle only non-temporal (also referred as static, non-time-based, and discrete) data types, such as text, images, fax, graphics, source codes, and binary codes, multimedia systems must also support temporal (similar terminology includes dynamic, time-based, and continuous) media types including video, animation, speech, and music. The term temporal media describes the temporal dimension of media such as video and audio, which contain sequences of data elements each having a position in time. For instance, a digital video clip contains a sequence of images called frames. Each frame must be played in a specific sequence in time and at a specific rate (typical frame rate is between 15 to 30 frames per second) to produce the correct visual effect. The timing constraints must be enforced during

capture and playback when temporal data are being presented to a human user. The success of conducting a multimedia session relies not only on the success of executing various programs and accessing numerous types of data files across the connected network, but also on the fulfillment of the timing requirements demanded by the temporary media types being utilized.

Different multimedia applications demand different communication requirements. A multimedia conferencing session, where data is presented once and then discarded, can be more tolerant to a higher error rate than an application that compresses and records an audio data stream for future playback. Although the multimedia conferencing session is less prone to transmission errors, it requires fast data delivery that is close to real-time. On the other hand, long transmission delays are of little concern to the audio recording application. Every element of the multimedia system must satisfy the requirements of the executing multimedia application and data streams it requested. Since multimedia systems are delay-sensitive, the multimedia services they employ must provide some kind of timing guarantees. Resource management, which is one of the major responsibilities of the multimedia network operating system (MNOS), must incorporate some form of resource allocation and scheduling schemes to map the requirements of the multimedia application onto the system and network capacity.

Due to the temporal nature of multimedia data types, the multimedia data streams that are transported across the computer network are significantly different from the asynchronous data traffic that typical local-area networks (LAN) are originally designed to support. Figure 1.1 lists some of the characteristics differences between multimedia data streams and asynchronous data.

Characteristics	*Asynchronous data*	*Multimedia Streams*
Load offered	Bursty	Long-lasting
Latency	Not critical	Low, time-critical
Jitter	Not critical	Low, less is better
Bandwidth	Highly bursty	Nearly constant & Predictable
Connection type	Connectionless	Connection-oriented

Figure 1.1: Differences between multimedia and asynchronous network traffic characteristics.

The duration for serving multimedia data streams is usually long-lasting (ranging from minutes, as in playing a musical file, to hours, as in playing a digital video) and requires high data rates (e.g., 1Mbits per second). On the other hand, asynchronous data transfers are typically short and bursty. A comparison of various data types with their required transmission bandwidth is given in Figure 1.2.

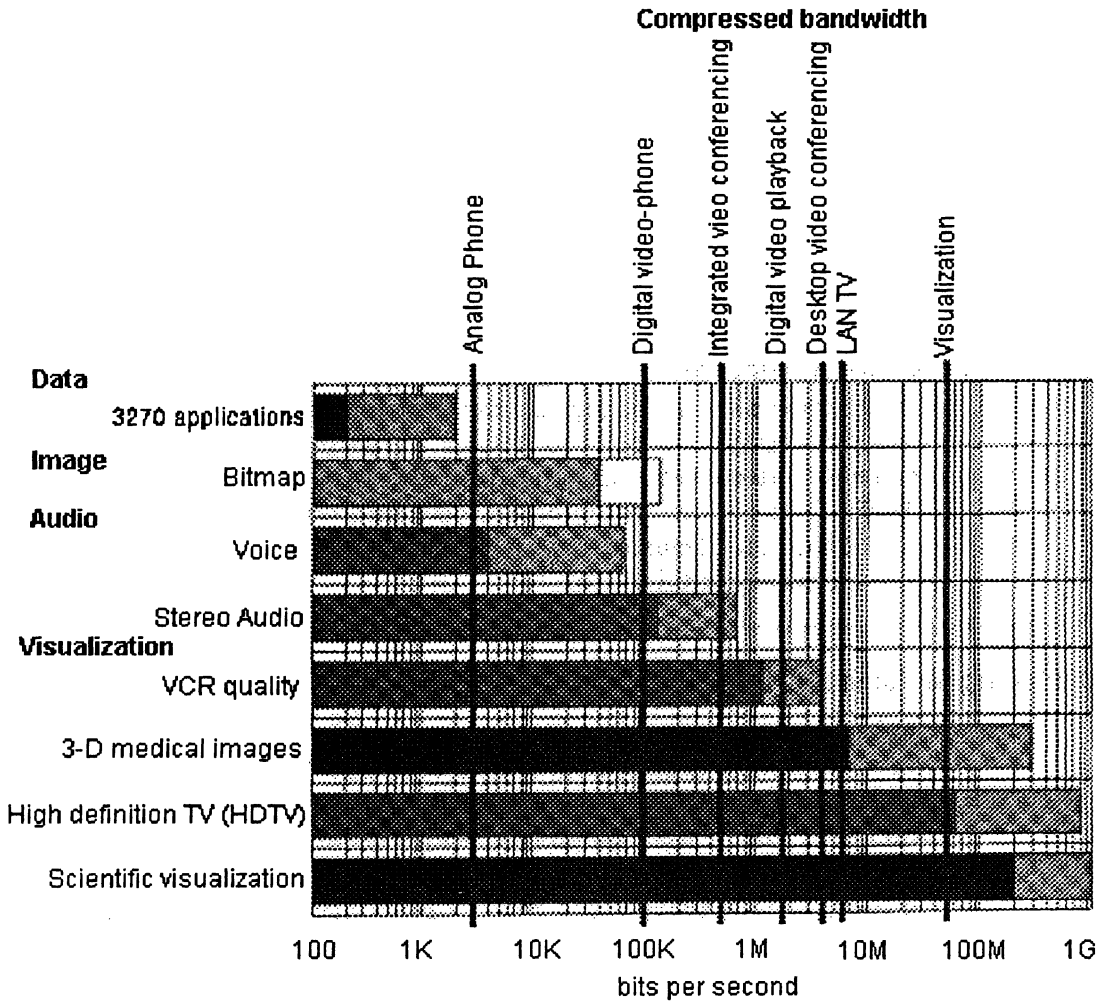

Figure 1.2: Bandwidth requirements of multimedia traffic (from [9]).

Asynchronous data transfers depend on the network to correct any transmission errors caused by data corruption or packet loss. Traditional transmission protocols can correct these errors by data correction schemes or by retransmission of the faulty data. In contrast, most multimedia applications can tolerate errors but demand on-time delivery of the required multimedia data

streams. However, they cannot tolerate the kind of delay that can be caused by data correction and retransmission. Instead, the corrupted data packets will be discarded. In the case of a digital video playback, a frame or two may be dropped without much distraction to the viewer.

Unlike asynchronous data transfers, where increased latency means longer delays in the transfer but otherwise harmless, latency may cause multimedia data to go completely out of synchronization; e.g., lip-synchronization during a video conference session, thus degrading the quality of the multimedia application from annoyance to unbearable. The inter-networking latency issue is illustrated in Figure 1.3.

Multimedia communication demands certain service guarantees from the computer network. These guarantees include per-session bandwidth requirements with limited latency and jitter that traditional networks are not designed to provide.

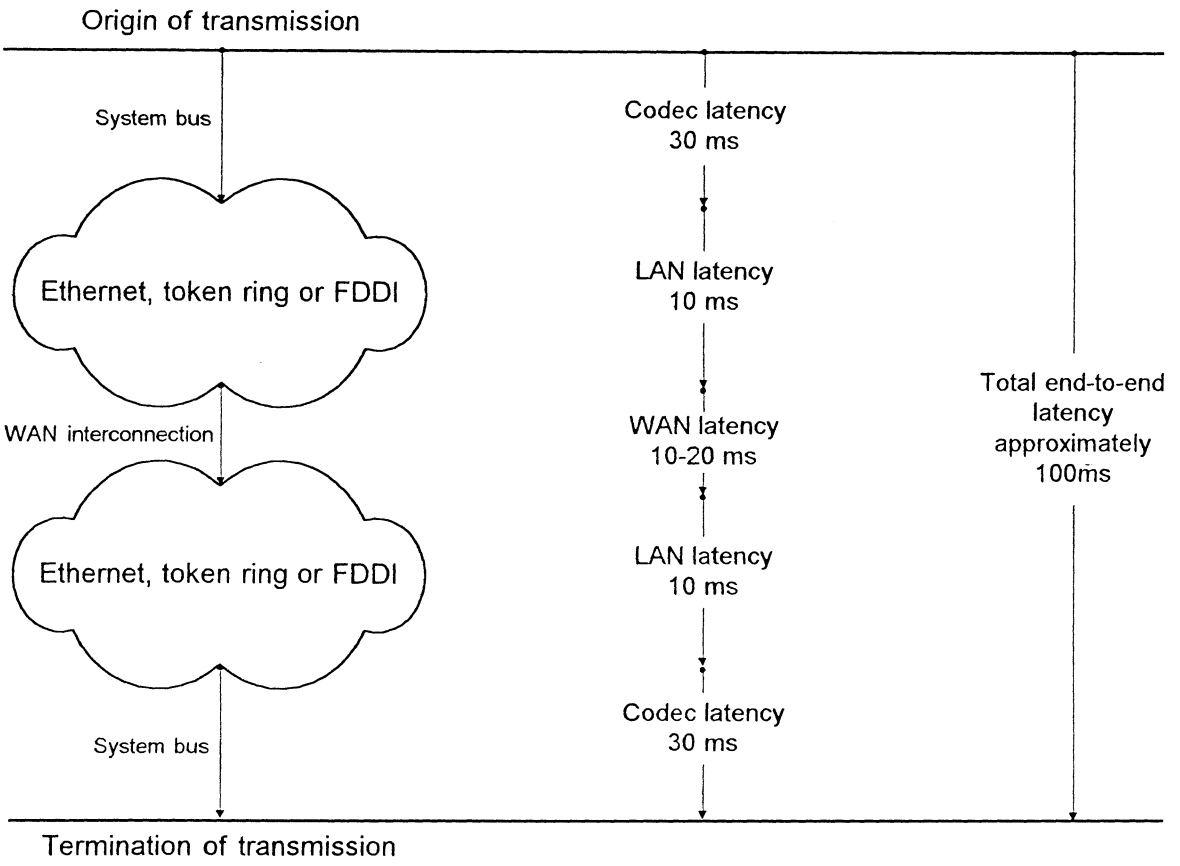

Figure 1.3: Latency concerns in inter-networking (from [10]).

1.2 Research Objectives

In this research, we classify the transmission and processing requirements of local and distributed multimedia applications by four major characteristics: (1) throughput, (2) local and global delays, (3) jitters, and (4) reliability. These

characteristics are known as Quality-Of-Service or QOS parameters. The multimedia network operating system must be based on a given QOS specification to perform resource allocation and management tasks. This process is similar to a workload request in a distributed computing system [11]. Existing literature on multimedia systems indicate that QOS definitions have not yet been standardized [12]. The main objective of this research is to identify the necessary QOS parameters for multimedia applications and the MNOS. The research also aims to propose an architectural model for mapping the multimedia application QOS requirements into the negotiation protocol of the networking system for resource allocations and utilization. The proposed model is designed to support multimedia applications with strict temporal requirements while isolating these applications from the details of network resource management, which include bandwidth allocation and process scheduling. The QOS Broker [13] model, suggested by Nahrstedt and Smith, is adopted here to negotiate and arrange for the delivery of end-to-end quality of service in our prototype distributed multimedia framework.

To make the research more complete, we have studied several synchronization techniques in multimedia systems and determined the synchronization schemes best suited to the proposed QOS model. The QOS negotiation model is simulated on a local area network with a single server and

four workstations. The server is running Windows NT Advance Server Operating System, version 3.51, while the workstations are running Windows NT Workstation Operating System, version 3.51. The entire network is running on IEEE 802.3 compliant 10-Base2 Ethernet. Finally, we analyze our simulation results to determine the effectiveness of the proposed QOS negotiation model when used in a multimedia networking environment.

1.3 Outline of the Dissertation

The remaining part of this dissertation is organized as follows: In Chapter 2, we provide some fundamental concepts in multimedia networking and discuss the need for a standardized QOS negotiation model to conduct multimedia network communication effectively. The concept of data topology models are introduced and the literature survey is presented.

Chapter 3 presents the different multimedia data types and defines the QOS parameters for multimedia applications. The multimedia application QOS parameters are then translated into network QOS requirements. We also analyze the network QOS requirements and discuss the issues related to skew, jitter, utilization, and data rate.

The QOS negotiation model is presented in Chapter 4. We first discuss the admission service for the transport subsystem and then describe the QOS negotiation process. A dynamic QOS negotiation scheme to accommodate the dynamic changes in network load conditions is also presented.

In Chapter 5, we discuss the computer implementation of our QOS negotiation model and some important issues in multithreaded programming. The experimental results are also discussed here.

Finally, the conclusion of this dissertation is given in Chapter 6. Future research directions pertaining to this research are also presented.

Chapter 2

Background and Related Work

2.1 Multimedia Data Types

Distributed multimedia systems deliver multiple sources of various spatial or temporal media types (also known as media objects) to compose mixed-media or compound documents [8]. QOS parameters can be derived from the methods used to compose these compound documents. As depicted in Figure 2.1, spatial composition links non-temporal media objects into a single entity or document. Spatial compositions must deal with object sizes, orientation, and placement within the document. Temporal composition arranges both temporal and non-temporal media objects according to their temporal relationships along a time-line as shown in Figure 2.2.

Temporal composition can be treated as a way of synchronizing multimedia objects. There are two types of temporal compositions: continuous synchronization and point synchronization. Lengthy multimedia presentations are best handled by continuous synchronization. An example would be video conferencing where audio and video signals are digitized at a remote site, transmitted over the network, then synchronized continuously at the receiving

station for proper playback. Video conferencing also demands full duplex communication which makes continuous synchronization a complex task. In contrast, point synchronization concatenates a single point of one media block (the starting point) to a single point of the previous media block (the endpoint). Slide shows with audio and voice annotations are good examples of point synchronization.

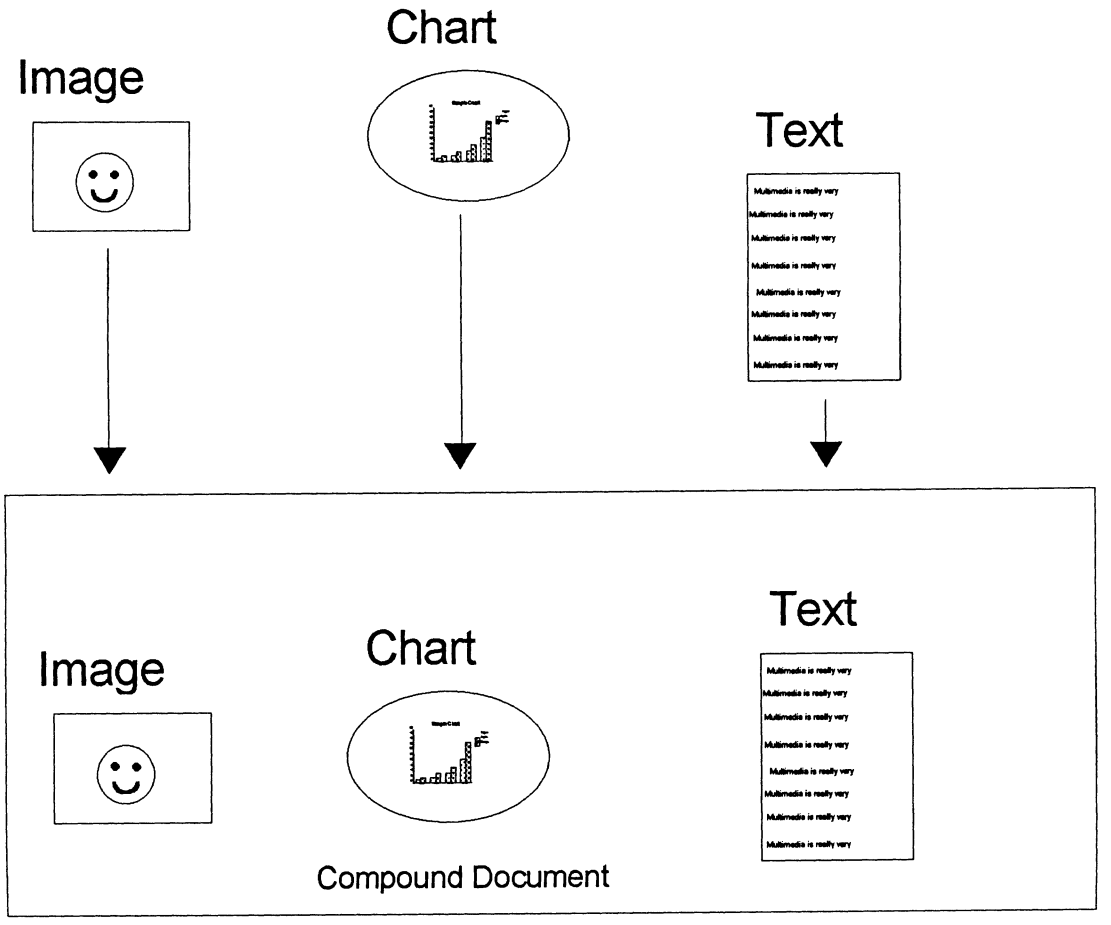

Figure 2.1: Spatial composition of non-temporal multimedia objects in a compound document.

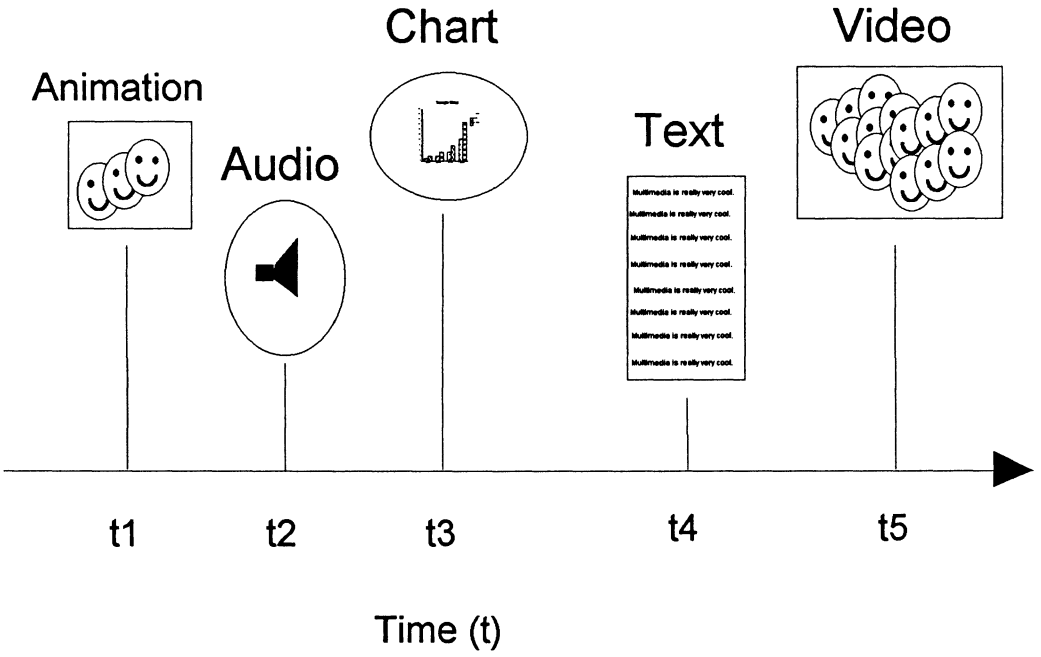

Figure 2.2: Temporal composition of multimedia objects according to their temporal relationships.

A multimedia system or a multimedia application is defined based on the number of media objects used in an application, the types of different media being supported, and the degree of media integration. The classification of multimedia systems based on these three criteria is illustrated in Figure 2.3.

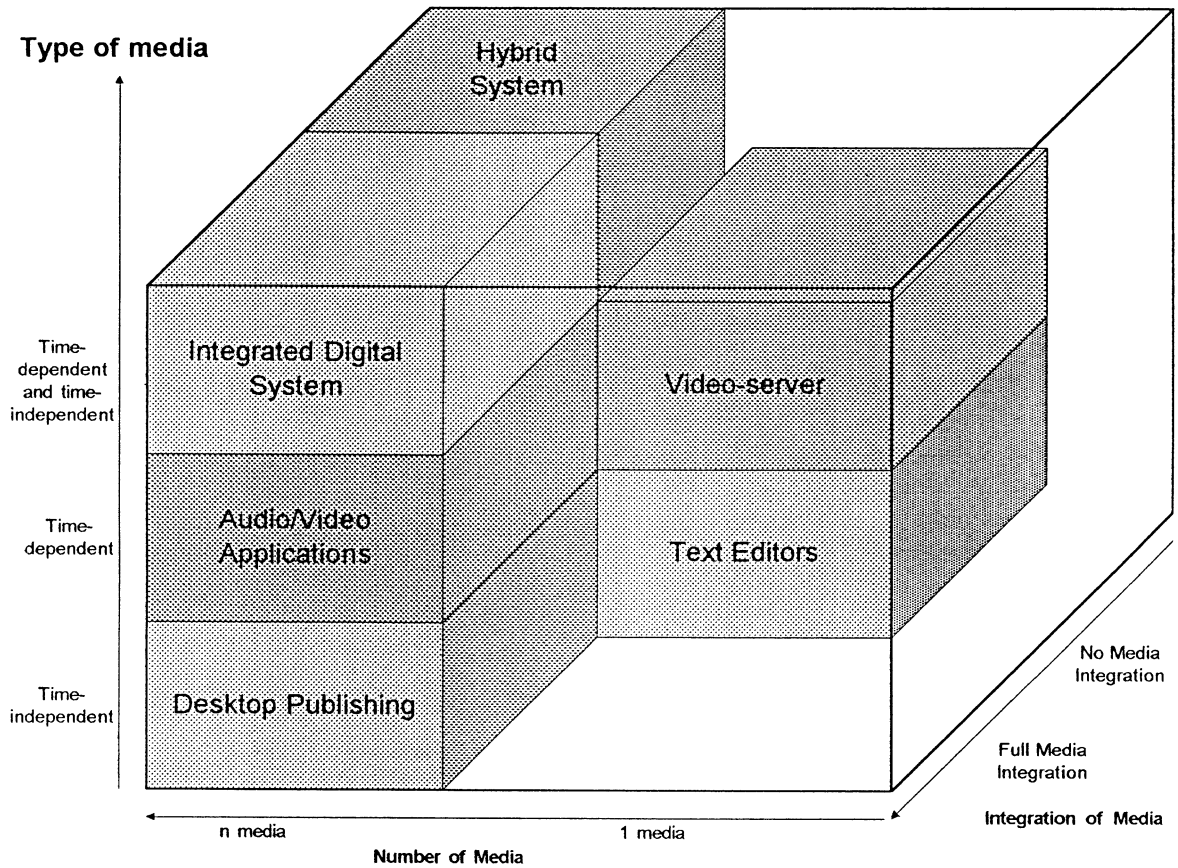

Figure 2.3: Classification of media utilization in multimedia systems and applications (from [14]).

2.2 Synchronization in Multimedia Networking

Synchronization is the mechanism that coordinates the order of events in the proper temporal sequence. Synchronization can be further classified as serial and parallel synchronization. Serial synchronization determines the rate at which

multimedia events occur within a single data stream. Serial synchronization is often referred as intramedia (or intrastream) synchronization since it ensures that QOS parameters, such as delays and jitters, of temporal media types are tightly bounded from the point of generation or retrieval to the point of delivery within a single data stream. When several temporal media streams are requested in parallel, potentially from different points of generation or retrieval, parallel (also referred as intermedia or interstream) synchronization is required to determine the relative timing relationships and schedule of multiple data streams. Either type of synchronization demands coordination between the multimedia application and the network resource managers in order to guarantee end-to-end synchronization. Overviews of multimedia synchronization methods can be found in Steinmetz [15] and Little et al. [16].

The implementation of a synchronization algorithm for a particular multimedia application requires a well defined set of QOS parameters for the underlying multimedia communications. Little et al. [17] and Butlerman et al. [18] suggested that the required synchronization algorithm can be built on data topology (location) models. As shown in Figure 2.4, four data topology models are suitable for multimedia systems:

1. Single source, local communication: A single media source produces and/or delivers the requested media streams to the playback devices

typically controlled by a PC workstation. Examples of the media sources may be a CD-ROM (stored media streams) or a digital video camera (live media streams). Examples of playback devices include a graphics video adapter and an audio interface with speakers. The communication link between the source, the workstation and the playback devices may be the system bus, a Small Computer Systems Interface (SCSI) channel, or an Asynchronous Transfer Mode (ATM) link. If the playback devices can maintain their proper playback speed, no other synchronization techniques are required. CD-ROM and video display adapter manufacturers often include on-board cache memory for buffering playback data streams in their products to ensure proper playback speed.

2. Multiple sources, local communication: Two or more media sources produce and/or deliver media streams to the local playback devices. A slide show with annotated speech and music fits into this type of data topology model. The workstation performing the slide show must perform media synchronization within the system.

3. Single source, distributed communication: A single media source produces and/or delivers media streams across a computer network to one or more nodes consisting of playback devices. Video-on-demand [19] is an

example of this model. It is assumed that there is no interactions between the clients and the server application. Synchronization becomes easier if all the playback devices can maintain proper playback speeds.

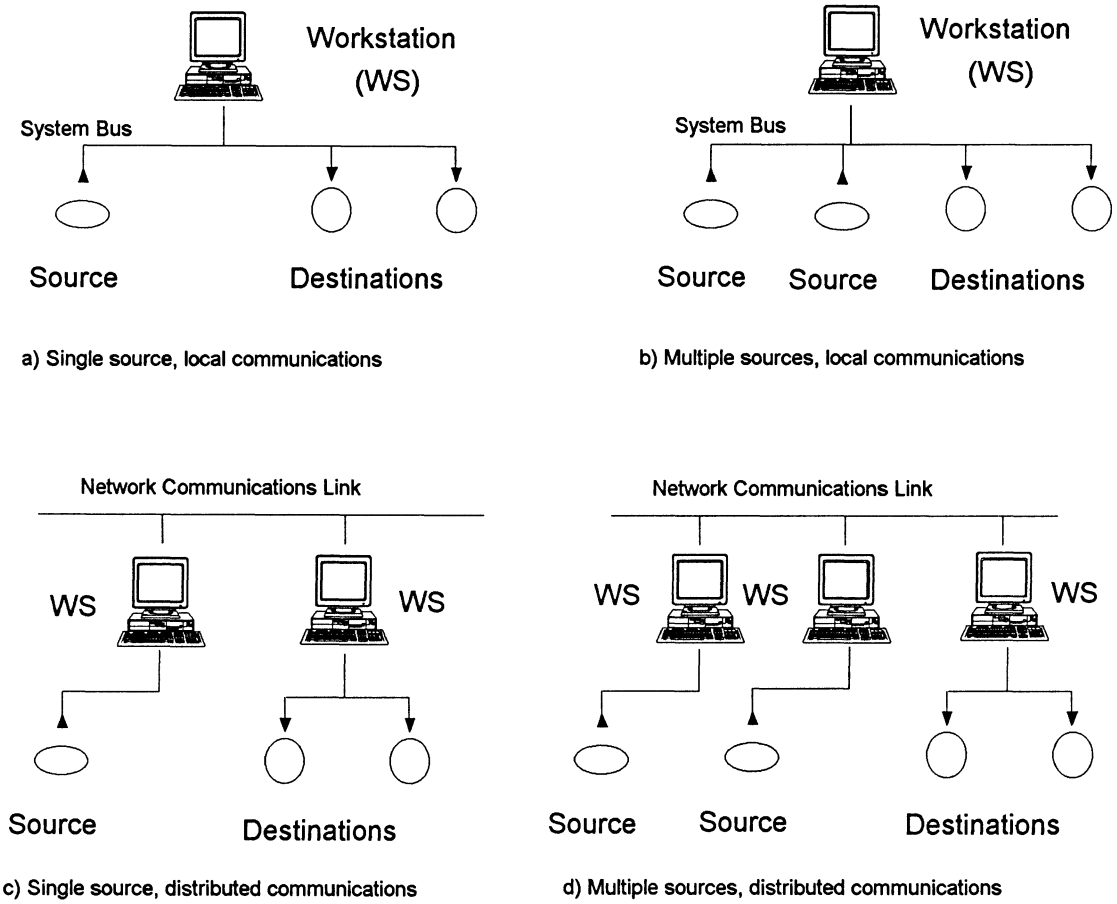

Figure 2.4: Data topology models for multimedia data streams: (a) Single source, local communication; (b) Multiple sources, local communication; (c) Single source, distributed communication; (d) Multiple sources, distributed communication.

4. Multiple sources, distributed communication: Two or more sources produce and/or deliver media streams to multiple playback devices located at different nodes distributed across the network. This complex model can be further broken down into three scenarios:

   i) Multiple sources located at a single node delivering media streams to one or more remote nodes. This scenario is similar to the "single source, distributed communication" model but with multiple sources of media streams.

   ii) Multiple sources from two or more different nodes delivering media streams to a single remote node.

   iii) Multiple sources from two or more nodes delivering media streams to two or more nodes. In this scenario, a workstation delivering media streams from its local media source can also be a recipient of incoming media streams generated from another remote node. Multimedia conferencing [20] for business workgroups [8] is a typical application of this model.

In order to implement an efficient synchronization algorithm for a specific application, one must efficiently translate the QOS parameters requested by the

multimedia application into the network QOS requirements. If the network QOS cannot guarantee service quality specified by the application QOS, a negotiation on QOS requirements must take place. If the network can support a lower set of QOS requirements while the application can tolerate the degradation of playback quality caused by lowering its QOS demands, a successful multimedia session can then be conducted. Although considerable research efforts have been devoted to the standardization of QOS parameters to guarantee quality end-to-end multimedia services, only a few studies focus on the translation of QOS requirements by an application into the negotiated network resource allocations. To this end, our research goal is to develop an architectural model to conduct these QOS translation and negotiation tasks.

2.3 Related Work

Tenet [21] is a research conducted by the University of California at Berkeley and the International Computer Science Institute. The Tenet approach to multimedia networking demands that any multimedia application should be able to request a level of network performance appropriate to its requirements. Tenet measures performance or QOS parameters including bandwidth (in terms of maximum packet size and inter-

packet arrival time), delay bounds, jitter bounds, and reliability bounds. These bounds may be statistical in nature. Since network performance cannot be achieved in the realm of unpredictable network application behavior, the interface between the network and the application is modeled as a contract to which both sides must comply.

The Desk Area Network (DAN) [22] project, developed at the University of Cambridge Computer Laboratory, employs a multimedia workstation equipped with a single ATM switch to interconnect multimedia peripherals (e.g., an audio interface and a video camera) with the system processors and memory. The internal architecture of a DAN workstation exhibits characteristics of an asymmetrical multiprocessor which differs from the typical symmetrical multiprocessor server design. By using an asymmetrical multiprocessor architecture, DAN's design objectives may exclude the low-level intrasystem communication demanded by a symmetrical multiprocessor, such as interprocessor communication, synchronization, and cache coherency. A high speed input/output (I/O) subsystem, such as SCSI, is utilized to interface mass storage devices with a DAN workstation. Since the storage devices are not completely integrated within the system, DAN is more flexible in terms of scalability as compared to traditional file servers. The primary memory model used is "stream caching" which delivers the streaming data from various

multimedia devices directly to the processor's secondary cache, thus bypassing the system main memory.

VuNet [23] is a high speed local area ATM network that can handle data rates at giga-bits per second. VuNet is implemented in the context of ViewStation, which is a distributed multimedia research conducted at the Massachusetts Institute of Technology (MIT). VuNet investigates the use of ATM technology to connect multimedia devices to a workstation without extending ATM into the workstation itself. It interconnects workstations and multimedia devices using a set of ATM links and switches. The objective of VuNet is to deliver real-time data such as video and audio from the network to the multimedia applications. A general overview of the VuNet project is shown in Figure 2.5.

Researchers at Washington University proposed using 3 by 3 ATM switches to connect a set of storage nodes to a high speed network [24]. Each storage node consists of SCSI channels and RAID (Redundant Arrays of Inexpensive Disks) devices to support massive storage and fault tolerance. A dedicated computer is attached to the ATM switches as a central manager to control the storage nodes. This project does not use ATM as the system interconnect bus but as the storage I/O back-bone network. The processors, system memory, and storage subsystems are still connect through a traditional system bus.

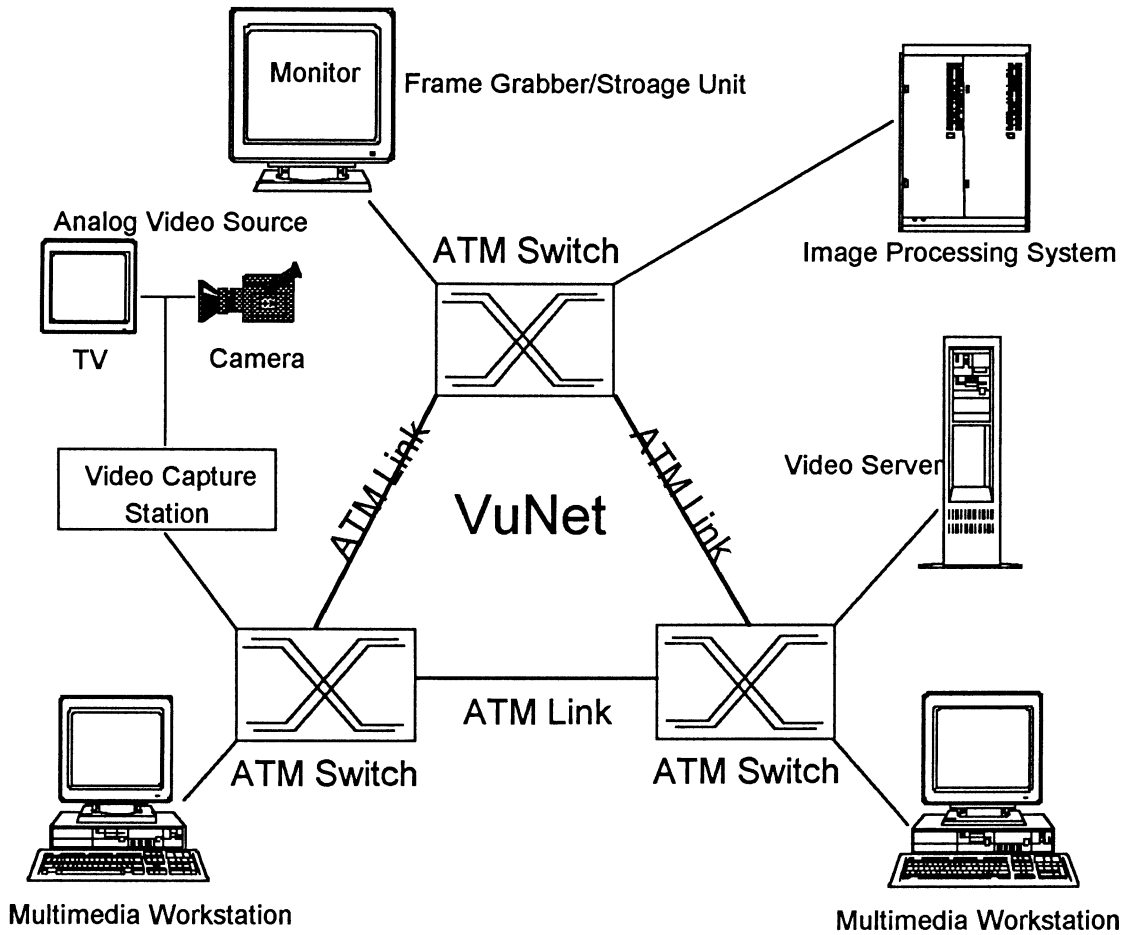

Figure 2.5: The VuNet high speed local area ATM network architecture designed to deliver real-time data such as video and audio to multimedia applications.

Project Athena [25, 26] is a multimedia computing project initiated at MIT to provide a flexible, efficient, and user-friendly prototype multimedia authoring

environment for creating distributed multimedia computing applications. AthenaMuse 1 [27, 28] is the platform-independent, multimedia authoring software environment derived from Project Athena. The latest AthenaMuse 2 [29, 30] converts the research agenda and experiences learned from AthenaMuse 1 into an extensible object-oriented software environment. The primary target platforms for this software system are UNIX-based workstations running the X-Windows System, a graphical user interface (GUI). Nevertheless, the software environment and generated applications will be portable across diverse hardware architectures and operating systems subject to the reasonable constraints of the target platform's hardware functionality.

The Multipoint Interactive Audio-Visual System (MIAS) [31] project funded by Esprit (Commission of the European Community) aims to study the necessary protocols and features required to support an efficient multipoint multimedia communication. The MIAS audio-visual terminal consists of dedicated hardware attached to an ISDN line (2B+D channels) incorporating a video input connector and H.261 codec to handle digital video data, while using a G.722 codec and associated audio I/O circuitry to handle digital audio data. The system runs under the Microsoft Windows platform using PC workstations. The PC workstations are also used for sharing as well as transferring data files, and performing conference control chores.

In this Chapter, we discussed the spatial and temporal properties of multimedia data types and the synchronization issues in delivering multimedia services. We also defined the four data topology models for multimedia networks that are used in this research. In the next Chapter we define the QOS requirements that multimedia applications impose on the underlying network. These requirements are expressed in a form of network performance criteria so that we can obtain a quantitative assessment of the services provided by the overall system.

Chapter 3

Quality of Service Requirements and Network Performances

3.1 Network Performance Criteria for Multimedia Applications

In distributed multimedia applications involving multiple sources and receivers, an intermedia synchronization scheme is needed to eliminate causes of asynchrony. Figure 3.1 shows a study conducted by Little et al. [17] regarding causes of asynchrony in a video telephony system. From this study the basic set of QOS parameters can be identified as speed ratio, utilization, average delay, jitter, bit error rate, and packet error rate. The above set of QOS parameters indicates that these parameters need to be derived from various multimedia devices and the distributed network. However, this is only appropriate for describing the connection quality that the network provides. The specified set reveals little information for describing the QOS requirements demanded by multimedia applications.

The first major goal of this research is to capture different media types and investigate their orientation, file size, temporal properties, and other features

specific to the media under investigation. Digital video is one of the major media types that we consider here. The Intel Smart Video Recorder Pro (ISVRP) video digitizing interface is used to digitize and store the video footage captured from a SONY Hi-8mm camcorder using the Y/C (S-Video) input to ensure image quality. The ISVRP interface is a second generation design based on the latest i750 video processor developed by Intel. Using the i750 video processor and Intel's Indeo 3.2 compression technique, the ISVRP interface is capable of capturing and compressing video in real-time up to 15 frames per second (fps) at low resolution. To successfully capture and then playback the digitized video with an acceptable quality, the application QOS parameters should include sampling size, sampling rate (or frame rate), playback delay due to system processing times, hard drive access times and video display adapter delays, and sample loss rate. Speech and music are also captured using the TurtleBeach MultiSound Monterey audio/sound adapter. For an acceptable digitized audio quality, the application QOS parameters must include sampling size, sampling rate, playback delay, and sample loss rate.

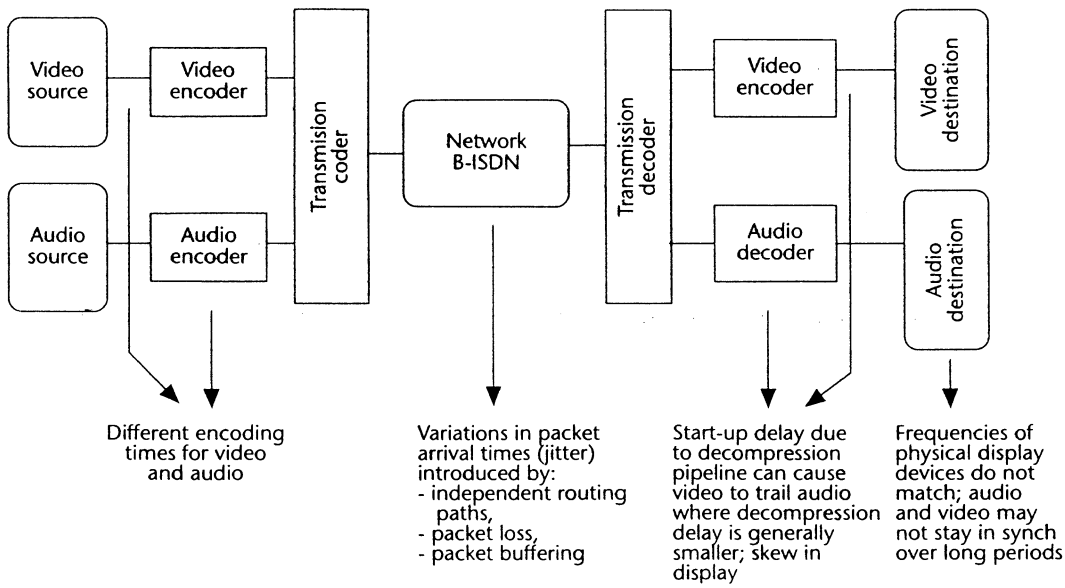

Figure 3.1: The study conducted by Little et al. regarding causes of asynchrony in a video telephony system (from [17]).

Although multimedia applications may have properties in common with other applications such as database systems, they have several specific features that directly affect the underlying network performances:

1. Multimedia applications require transmission of continuous media types in either real-time or near real-time.

2. The amount of data to be transmitted is substantial and the processing power involved in encoding and decoding the continuous media data may be considerable.
3. Many multimedia applications are distribution oriented; e.g., video on-demand, where streams of multimedia data need to be distributed to many users at the same time.

From these observations, we have chosen four performance criteria to characterize the network behavior when handling continuous media traffic. These criteria are actually used in specifying the QOS for our multimedia communications model:

1. Throughput: This is the transfer rate or data rate between two communicating end-systems. Throughput is also known as the bandwidth of the network.
2. Transmission delay: This is the delay for a block of data to propagate from one end-system, via a network, to another end-system. Transmission delay is often called the network latency.
3. Delay variations: Data transmissions can be affected by the physical properties of the network, such as data corruption, crosstalk between cables, and network overloading. These conditions all contribute to

variations in the transmission delay of the network. A term often used to describe these variations is jitter.

4. Error rates: Error rates are measurements of the reliability, or the resilience to errors, of the network. Errors can be caused by data loss, data duplication, data alteration, or incorrect-order delivery of data packets.

3.2 Application QOS Parameters

For multimedia computing, the required QOS depends on the individual multimedia application and the media objects being served. One of our research tasks is to provide a general framework to parameterize the QOS specifications for each media object for multimedia applications. The QOS specifications for a media object include quantified values to describe the necessary playback quality and the playback accuracy. For a time-dependent media object which contains a sequence of time-dependent information units, also known as Logical Data Units (LDUs), the accuracy in timing during playback depends on whether the QOS specification can be satisfied by the underlying network or not. Figure 3.2 shows an example of the LDU hierarchy of a digital animation which requires lip-synchronization.

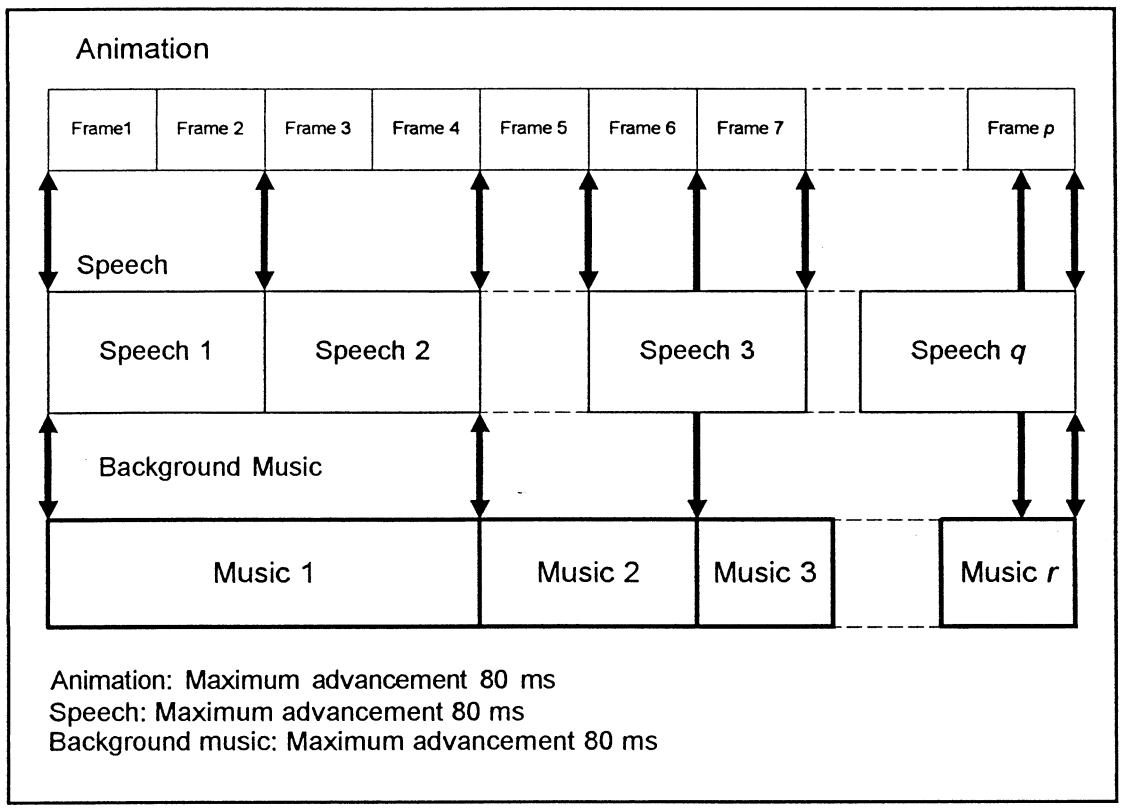

Figure 3.2: The LDU hierarchy of a digital animation which requires lip-synchronization.

Depending on the type of multimedia application, absolute synchronization requirements can be relaxed to various degrees for each media object without adversely affecting their presentation quality. We classify the QOS specifications for a media object based on three major characteristics:

1. Presentation qualities that are independent of temporal relationships.
2. Presentation qualities that are affected by temporal relations among other media objects.

3. Quality degradation caused by the presentation environment.

Color depth and resolution are the two time-independent QOS parameters that specify presentation quality for still pictures and digital video. For digital audio data, color depth is replaced by the number of audio channels and resolution is replaced by the sample size of the digital audio file. Apparently an audio file with mono sounds and a small sample size demands less bandwidth than the one with stereo sounds and a moderate sample size to be transmitted across the network. In other words, these QOS parameters directly affect the throughput of the network. We shall derive an approach to map the QOS parameters into network QOS requirements in Section 3.3.

The time-dependent QOS parameters of digital video include the frame rate which determines the smoothness of motion during playback. It is generally agreed that at least 15 fps is required to reproduce fluid motion in a video sequence. Some video conferencing applications can perform satisfactorily between 11 fps to 14 fps due to the limited amount of movements typically involved in this kind of application. In the case of digital audio, frame rate is replaced by the sample rate.

For aperiodic data (such as still images and text) time-dependent QOS parameters include preferred end-to-end delay, acceptable end-to-end delay, and

unacceptable end-to-end delay as measured with respect to real-time or with respect to other aperiodic data to be presented[32]. An example of end-to-end delay is illustrated in Figure 3.3. These end-to-end delay parameters are also applicable to periodic data such as video and audio. However, instantaneous delay variations or jitter can seriously affect the synchronization between periodic data streams. Therefore, maximum acceptable jitter is included in our application QOS parameters to define an affordable synchronization boundary for the instantaneous difference between two synchronized data streams.

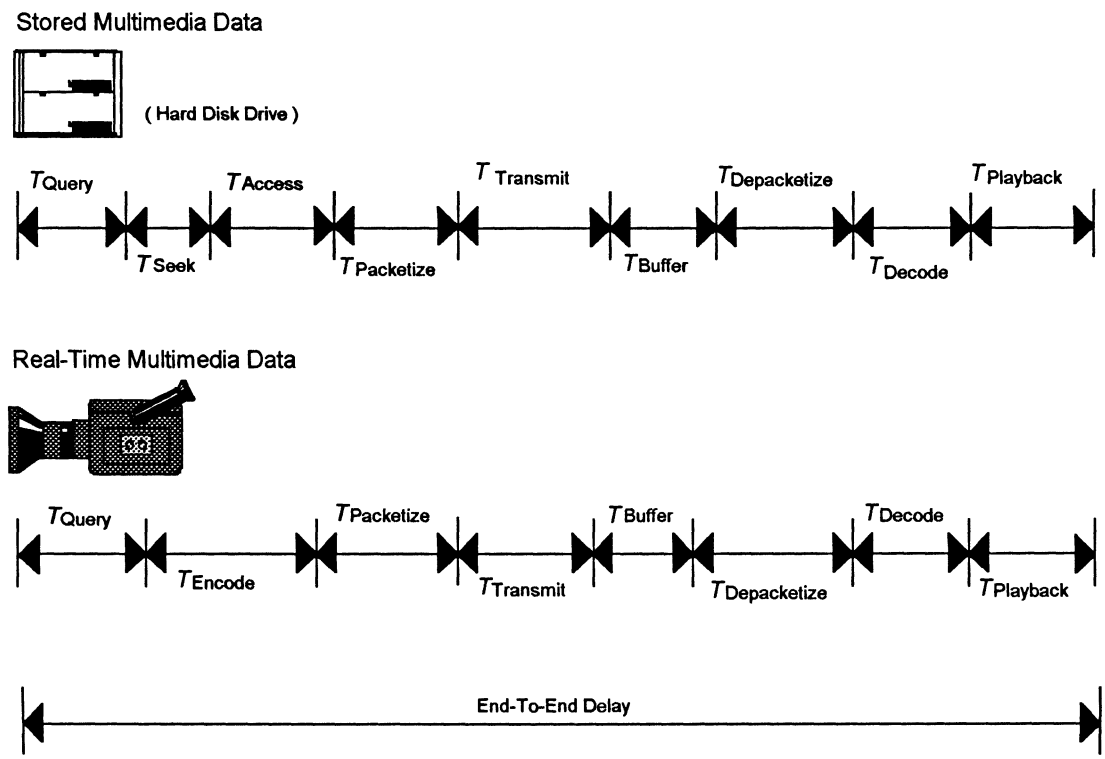

Figure 3.3: The end-to-end delay of a distributed multimedia comprises all the delays experienced at the source site, the computer network, and the receiver site.

Delay variations are contributed by sample losses during transmission and the reliability of the underlying network. To quantify these variations, we have included the maximum sample loss rate (SLR), maximum bit error rate (BER), and the maximum packet error rate (PER) as the acceptable QOS parameters for a media object. These parameters can be used to describe time skew with respect to real time or with respect to periodic data streams. The average difference in presentation times over some n synchronization intervals between two corresponding media objects is called skew. For media objects such as video and audio, data can be lost during playback resulting in dropped frames or gaps in the presentation. Such losses cause data streams to advance in time. This synchronization problem is called stream lead. On the other hand, duplicating a data frame causes the data stream to retard in time or a stream lead. Figure 3.4 illustrates the effects of time skew. Figure 3.5 shows how jitter is corrected by either dropping or duplicating frames of data and skew may be corrected by dropping data frames.

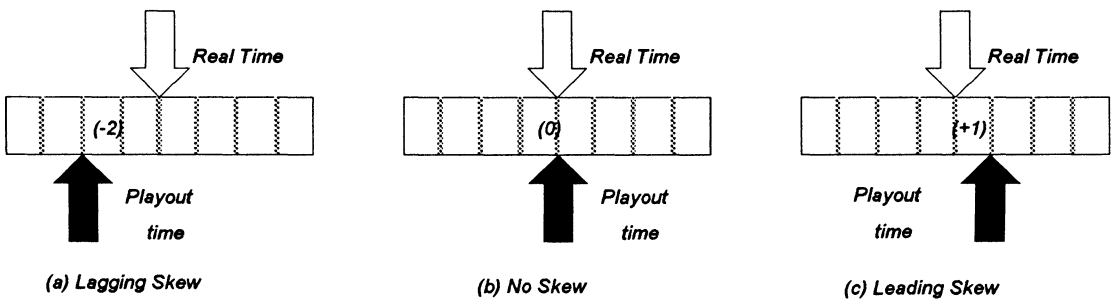

Figure 3.4: Effects of time skew.

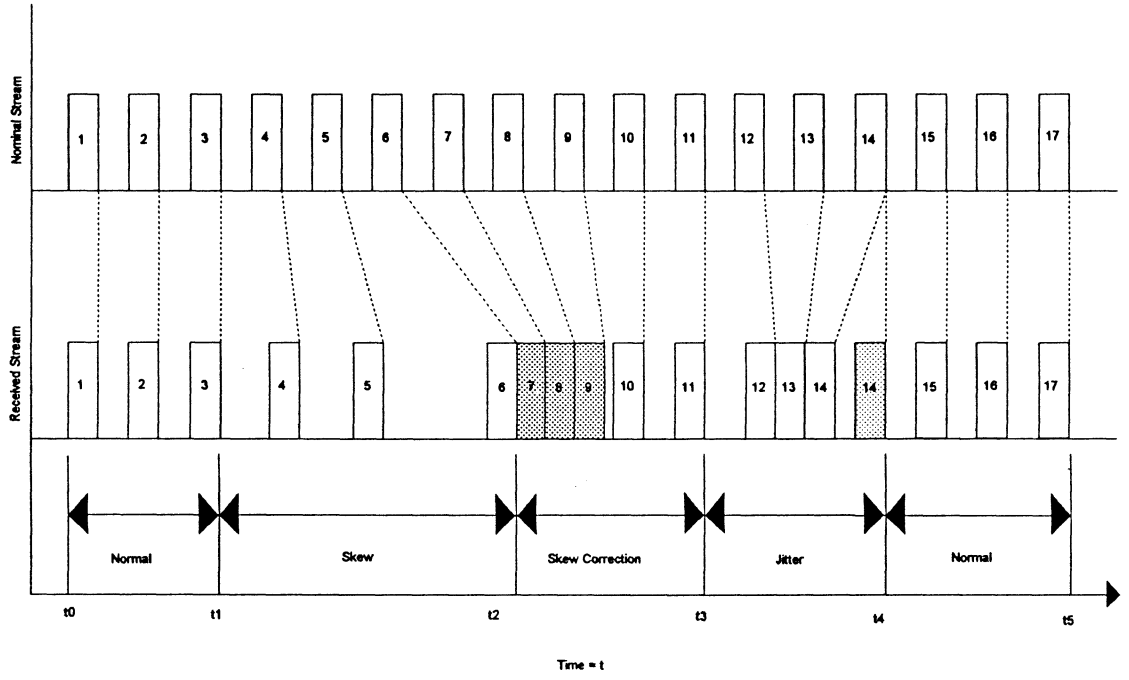

Figure 3.5: A nominal multimedia data stream as compared to the received data stream with errors caused by skew and jitter of the original data during transmission (from [33]).

To account for time skew, we need to address the synchronization problem both during the production of multimedia data objects and during the presentation of these objects. In order to guarantee the QOS required by a multimedia application, all involved media objects (e.g., video and audio) are captured, recorded, and edited with no skew during production. This is known as the production-level synchronization and it ensures that all media objects are *"in- sync"* prior to the presentation of the data at the user interface. Unlike production-level synchronization, which can be controlled during production,

presentation-level synchronization depends on the networking system conditions during presentation. The presentation-level synchronization defines the boundary for an acceptable presentation quality as perceived by the human user at the workstation end. The degree of acceptable presentation quality is expressed as the QOS parameters defined for maximum lead-skew and maximum lag-skew. The actual values for these parameters are found by exhaustive experiments as well as derived from literature in this research area. Figure 3.6 depicts some of the QOS values appropriate for presentation-level synchronization [34].

The last application QOS parameter we need to define is the priority level that supports multithreaded programming. Multithreading is a form of concurrent programming in a multitasking environment. Here, priority is defined for the relative importance among the different media objects that are managed by an application. Consider the case where a real-time application shares the same system resources with non real-time programs. The real-time application, while waiting for some continuous data packets, must compete for system resources with non real-time applications which need only asynchronous data. Since continuous data packets have deadlines to meet, they should be assigned with a higher priority than asynchronous data packets that have no timing restrictions. There are four priority levels: real-time, high, normal, and idle.

Priority is not an option that can be selected by the user. It is defined by the application and the nature of the multimedia objects it handles. Details on priority and multithreading are given in Chapter 5 when we discuss the implementation of our QOS model. We have now covered the set of application QOS parameters defined for media objects and a summary is shown in Figure 3.7.

Media		Mode, Application	Quality of Service
Video	Animation	Correlated	+/- 120 ms
	Audio	Lip Synchronization	+/- 80 ms
	Image	Overlay	+/- 240 ms
		Non-overlay	+/- 500 ms
	Text	Overlay	+/- 240 ms
		Non-overlay	+/- 500 ms
Audio	Animation	Event Correlation (e.g., dancing)	+/- 80 ms
	Audio	Tightly Coupled (stereo)	+/- 11 µs
		Loosely Coupled (dialogue mode with various participants)	+/- 120 ms
		Loosely Coupled (e.g., background music)	+/- 500 ms
	Image	Tightly Coupled (e.g., music with notes)	+/- 5 ms
		Loosely Coupled (e.g. slide show)	+/- 500 ms
	Text	Text Annotation	+/- 240 ms
	Pointer	Audio Related to the Item to Which the Pointer Points	- 500 ms, + 750 ms[a]

a. Pointer prior to audio for 500 ms; audio prior to pointer for 750 ms.

Figure 3.6: Quality of Service for presentation synchronization purposes (from [34]).

Media Object	Graphics and Bitmap Image	Digital Video	Digital Audio
Quality of Service	Color Depth	Color Depth	Audio Channels
	Resolution	Resolution	Sample size
		Frame Rate	Sample Rate
		Max. Jitter	Max. Jitter
		Max. Lead Skew	Max. Lead Skew
		Max. Lag Skew	Max. Lag Skew
		Frame Loss Rate	Sample Loss Rate
		Max. Bit Error Rate	Max. Bit Error Rate
		Max. Packet Error Rate	Max. Packet Error Rate
		Priority	Priority

Figure 3.7: A set of application QOS parameters defined for media objects.

3.3 Translating Application QOS Parameters into Network QOS requirements

To provide multimedia applications with end-to-end service guarantees, some kind of network resource management scheme must be incorporated into

the application, the networking system, and the operating system at the endpoints as well as between the endpoints and the network. The application QOS parameters defined in the previous section allow the multimedia application to submit a request for QOS guarantees, at the application's perspective, to the networking system for performing resource allocation. Resource allocation and management are complex tasks which will be discussed in the next Chapter when we describe our QOS negotiation model. However, before any resource allocation can begin, the networking system must understand the types of service guarantees that the application really needs by interpreting the set of submitted application QOS parameters. This is achieved by translating the application QOS parameters into the network QOS requirements.

Our model specifies network resources as network QOS requirements based on three domains that govern the connection quality of the network. The first domain involves throughput specifications which include packet size and packet rate. The second domain deals with network traffic specifications. Here, we have included preferred end-to-end delay, acceptable end-to-end delay, unacceptable end-to-end delay, interarrival rate, and packet loss rate. The third domain is synchronization specifications which include network jitter, lead-skew, lag-skew, BER, PER, and priority.

The translation between application QOS parameters and network QOS requirements is done by several mapping functions. The network packet size S_N is a known value determined by the transport subsystem (e.g., TCP/IP running on 10-Base2 Ethernet) used to implement the network. The network packet rate R_N is determined by

$$R_N = (\lceil S_A/S_N \rceil) * R_A \qquad (4.1)$$

where S_N is the packet size of the network and it is predetermined by the network itself, S_A is the sample size of the application media object, R_A is the media object sample rate, and ($\lceil\ \rceil$) is the ceiling function. The subscript 'A' denotes that the value is defined for an application QOS parameter whereas the subscript 'N' denotes that the value is defined for a network QOS parameter. The sample size of a still image or a video frame is determined by the resolution and the color depth of the image. For example, a video frame with resolution of 160 pixels by 120 pixels and a color depth of 8 bits per pixel (i.e., 256 colors) has a sample size S_A of 160 * 120 * 8 = 153,600 bits or 19,200 bytes. If the video clip is to be played back at 15 fps, then R_A = 15 fps. Assuming that the network packet size S_N is 64 bytes, the required network rate R_N would be:

R_N = $\lceil$ (19,200 bytes per frame/ 64 bytes per packet)$\rceil$ * 15 fps

= 4,500 packets per second

The interarrival rate I_N is calculated as

$$I_N = (1/R_A) \ / \ (\lceil S_A/S_N \rceil) \tag{4.2}$$

The calculations for the network unacceptable end-to-end delay $C_{N_{una}}$, acceptable end-to-end delay $C_{N_{acc}}$ and preferred end-to-end delays $C_{N_{pre}}$ are expressed by the following equations:

$$C_{N_{una}} = (C_{A_{una}} - RT_A - WT_A) \tag{4.3}$$

$$C_{N_{acc}} = (C_{A_{acc}} - RT_A - WT_A) \tag{4.4}$$

$$C_{N_{pre}} = (C_{A_{pre}} - RT_A - WT_A) \tag{4.5}$$

where RT_A and WT_A are the processing times for the application to read data from a remote source and to write data to a local buffer, respectively. These processing times are measured in advance and exchanged during the QOS negotiation process.

The rest of the network QOS requirements are directly mapped onto the given application QOS parameters. They are:

Packet loss rate L_N = Sample loss rate L_A (4.6)

Network jitter J_N = Maximum acceptable jitter J_A (4.7)

Lead-skew SD_N = Maximum lead-skew SD_A (4.8)

Lag-skew SG_N = Maximum lag-skew SG_A (4.9)

Network bit error rate BER_N = Maximum acceptable bit error rate BER_A (4.10)

Network packet error rate BER_N = Maximum packet error rate BER_A (4.11)

Priority set for application P_A = Priority set for network P_N (4.12)

In this Chapter we have discussed the behavior of continuous media traffic and the four performance criteria which are critical for the communication subnetwork to support multimedia applications. Based on these criteria (i.e., throughput, transmission delay, delay variations, and error rates), we have derived the basic set of QOS parameters for multimedia applications. These application QOS parameters are then translated into network QOS requirements based on the mapping functions (equations 4.1 through 4.12). Although these mapping functions may be straight forward to obtain, the entire translation process is in fact a complex procedure due to the dynamic nature of the networking environment. Recall the four data topologies outlined in Chapter 2. Media streams can come from several different sources, go through different parts of the network, and be delivered to different users (or sinks) thus making the translation process a difficult one. If parts of the network resources are not

available at the time of the QOS request, the network must decide to either reject or attempt to accommodate the request by modifying the QOS requirements to some levels that are acceptable by both the application and the network. The details of this QOS negotiation process are given in the next Chapter.

Chapter 4

The QOS Orchestration Model

4.1 Integrated QOS Orchestration Architecture

To satisfy the QOS requirements demanded by multimedia applications, network resource management alone is inadequate [35]. There have been debates on placing the responsibilities of these QOS requirements to the application level software instead of redesigning a whole new set of multimedia networking protocols. As a result, much of the current multimedia applications have to incorporate proprietary codes to maintain isochronous transmission for media objects since most commercial networking operating systems are not designed to support continuous media. The main disadvantages of this brute force approach are:

i) Distributed multimedia applications are difficult to develop without standard support for continuous media communication from the underlying operating system.

ii) Proprietary QOS management imposed by different multimedia applications may result in poor performance when these applications are executing together in a multitasking environment.

iii) There will be some wasted bandwidth and processing power across the network since there is no integrated resource management overseeing all the distributed resources.

This analysis suggests that there is a need to manage resources among the application, network, operating system at the communicating end-station, as well as between the communicating end-stations and the network itself in a unified and balanced manner. We, therefore, advocate the notion of an integrated QOS architecture [36], whereby application QOS requirements should be mapped through all the layers of the entire system. Our proposed endpoint communication model, which is designed to support an integrated architecture, embodies two major components: an application subsystem and a transport subsystem as shown in Figure 4.1. The application subsystem provides facilities such as application QOS management, multimedia service management, I/O device management, media object synchronization, and media data delivery to the application. End-to-end connection management, data flow and throughput control, data packet ordering, and data flow management at the network interface are all functions provided by the transport subsystem. Each subsystem is further divided into functional layers similar to the Lancaster distributed multimedia architecture [37].

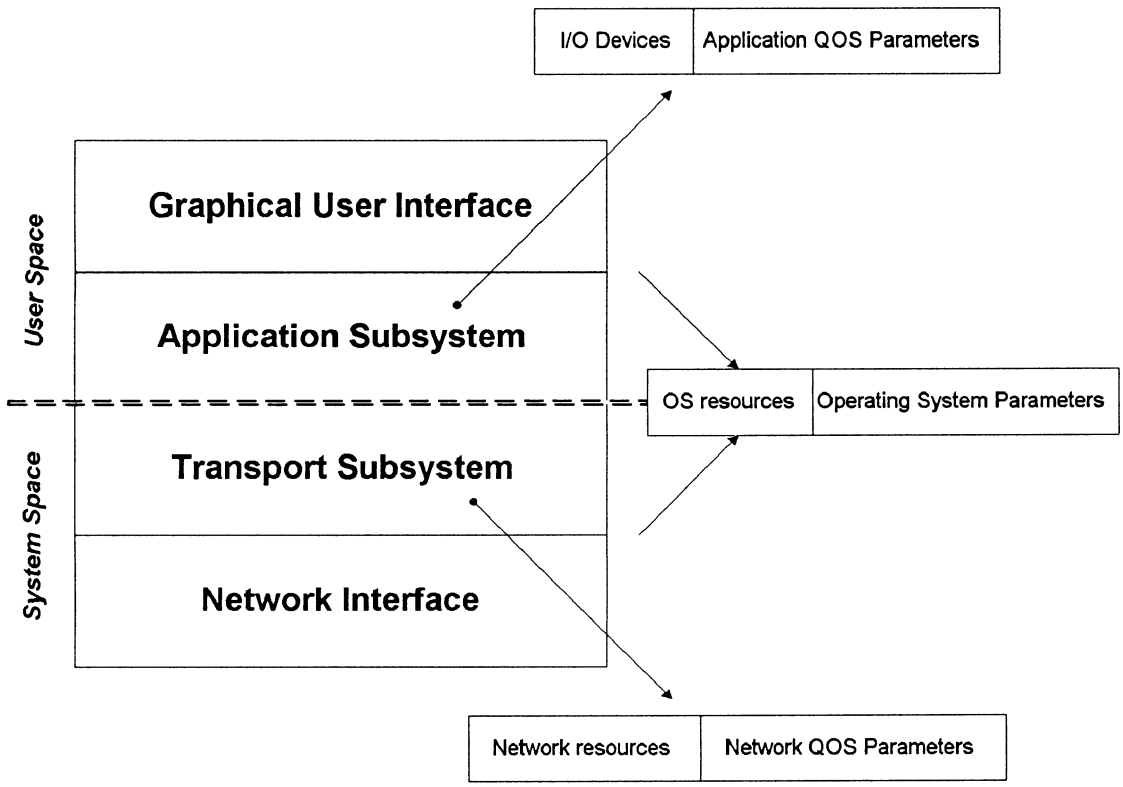

Figure 4.1: The proposed endpoint communication model embodies two major components: an application subsystem and a transport subsystem (from[13]).

In our model, resource management is conducted over the entire network architectural layers, from the distributed application layer down to the network layer as depicted in Figure 4.2. The QOS negotiation agent is at the interface between the application subsystem and transport subsystem. It is an entity that orchestrates required resources for the multimedia application. The orchestration services include end-to-end QOS negotiation, renegotiation, QOS degradation detection, and QOS coordination over multiple related connections.

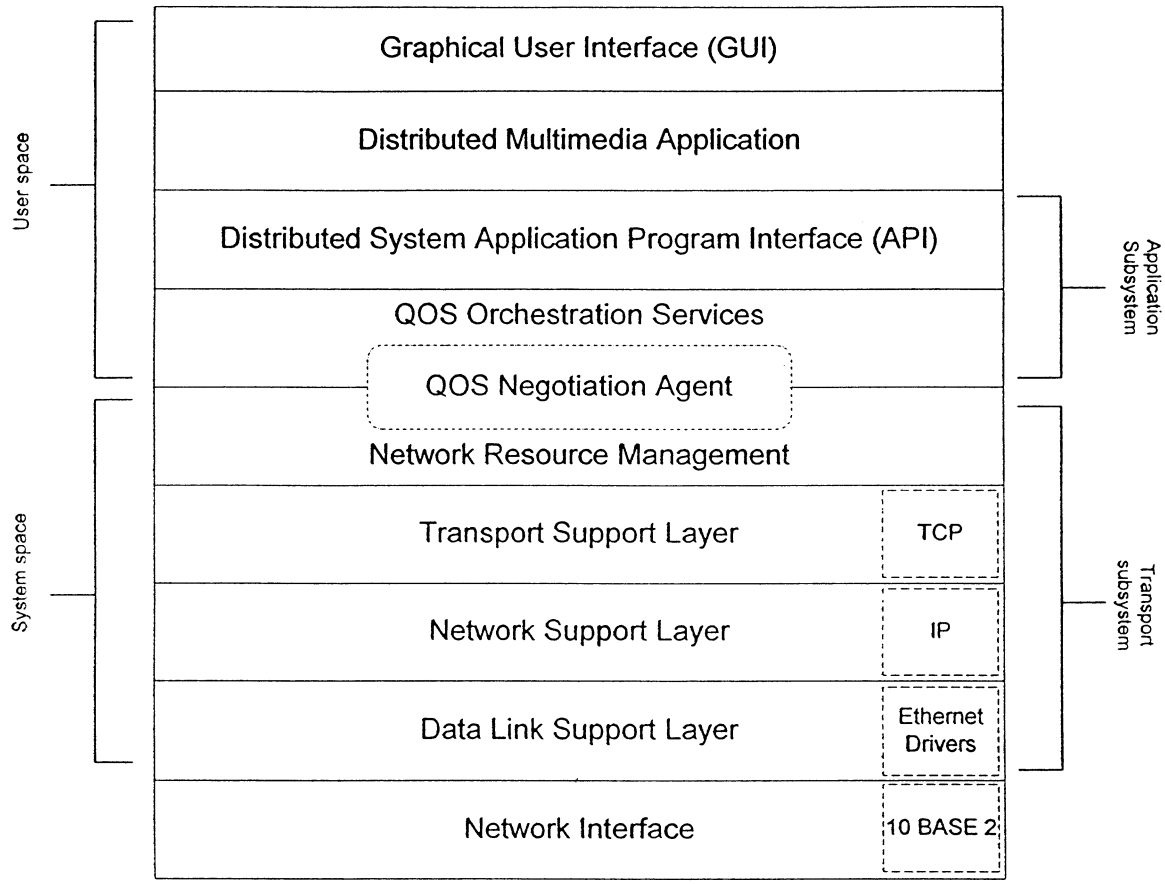

Figure 4.2: The layered architecture of our integrated QOS orchestration model.

All multimedia application resources are treated as local or remote I/O devices that can carry continuous media traffic. The resources are parameterized through the translation between application QOS parameters and network QOS requirements as discussed in Chapter 3. For simplicity, we decide to allocate each media object its own virtual I/O channel or virtual circuit in the transport layer. One may argue that continuous synchronization can be achieved by multiplexing

the required media objects into a single virtual circuit in the proper throughput ratios. Our experiments indicate that a more general solution of separating media objects, each with its own virtual circuit, has a number of advantages over the multiplexing approach [38]:

(i) Implementation is simplified with a single media per virtual circuit approach. Comparatively, the complexity and processing overhead to multiplex and demultiplex are substantial.

(ii) Most often media data are compressed to improve throughput during transmission. Compression schemes vary greatly among different media types. With compressed data, multiplexing and demultiplexing different media objects over a single virtual circuit may lead to excessive end-to-end delays.

(iii) Multiplexing is not feasible when several requested media objects are originated from different sources.

(iv) Separate virtual circuits may be processed in parallel to increase performance.

(v) Separate virtual circuits allow the use of best-suited I/O channels for individual media objects, thus leading to better resource utilization. Contrarily, multiplexing leads to compromise QOS and poor utilization of the I/O channels.

In addition, our transport subsystem requires only unidirectional (simplex) virtual circuits for transporting continuous media. The argument is that most continuous media are inherently unidirectional in nature; e.g., video on-demand delivers digital video to an end-station in one direction. As resources must be explicitly reserved to provide QOS guarantees, network bandwidth will be wasted on supporting duplex virtual circuits when only unidirectional transfers are needed. For situations where duplex communication is required, two simplex virtual circuits are employed. An added advantage for using two simplex virtual circuits instead of a single duplex virtual circuit is that the QOS requirements of the two directions are generally different and should be handled separately.

4.2 The QOS Negotiation Agent

In a QOS-based resource management scheme, it is not sufficient to specify only a QOS level, the protocol profile, plus the service class at session creation time and assume that all conditions will statistically remain intact for the life of the session. It is more appropriate to adopt a dynamic QOS control scheme [39, 40, 41] since QOS requirements and network traffic frequently change during the course of a single session. Based on this observation, our QOS negotiation agent is designed to maintain the required services even when system condition changes during a multimedia session.

The QOS negotiation agent relies on the supplier-consumer paradigm, where the consumer requests a service, or a product, from the supplier and the supplier delivers the service if the consumer agrees on the price. A deal may not work out if either the consumer is not satisfied with the given product or the supplier is not pleased with the offered price. A third party, an agent, may be called upon to mediate between the consumer and the supplier in order to close a deal. Using dynamic QOS control, the QOS negotiation agent plays the role of a mediator in helping the consumer and the supplier to come to an agreement for multimedia services. The consumer, in this case, is a human user, or a computer program, who requests multimedia services by executing some application. The commodities being consumed here are the media objets requested by the consumer. The terms for this deal to close depend on the availability of the resources that can be allocated for processing and transmitting the media objects. Naturally, the supplier is the remote application that manages the requested multimedia resources at a remote site; e.g., a video database server for video on-demand services.

A multimedia session begins when the consumer calls for services at the graphic user interface (GUI). The consumer application creates a QOS profile registry (database) and stores a set of application QOS parameters as described in the previous Chapter. According to the QOS values, the QOS negotiation agent

begins to allocate local resources at the QOS orchestration layer. Through the transport subsystem, this agent also gathers resource allocation information from the network resource management as well as from the supplier QOS negotiation agent at the remote site. Upon some request for service, the remote supplier initiates QOS orchestration similar to the process performed by the consumer application. The supplier QOS agent determines the availability of resources for establishing the specified connections. If all the QOS requirements can be satisfied at the time of the request, the service is admitted with an end-to-end QOS guarantee. Figure 4.3 shows the basic steps to establish a connection involving the consumer, the QOS negotiation agents, and the supplier. An example of a successful QOS negotiation is shown in Figure 4.4.

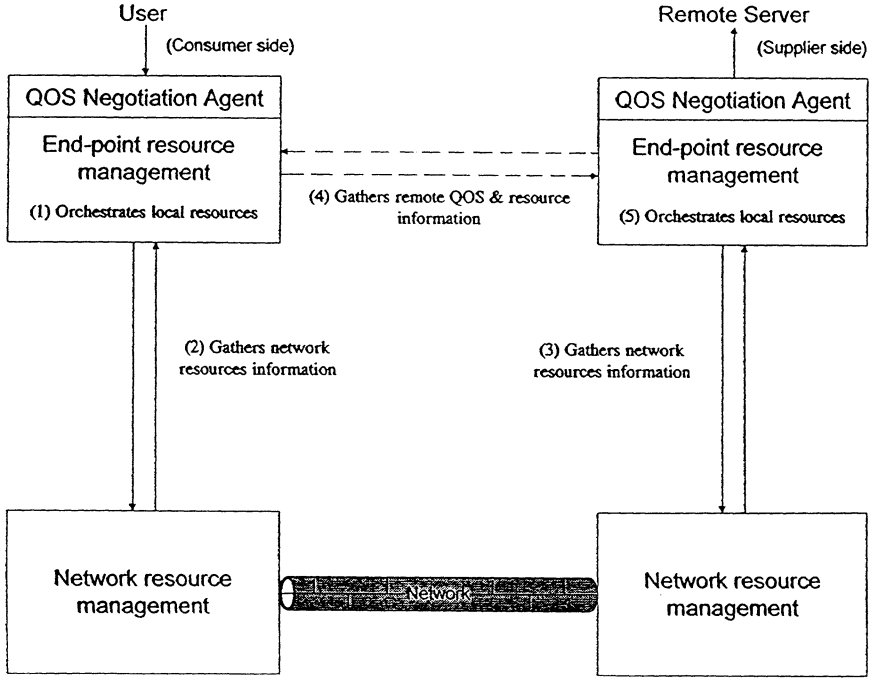

Figure 4.3: The basic steps in establishing a connection from the consumer to the supplier with the help of the QOS negotiation agents on each side.

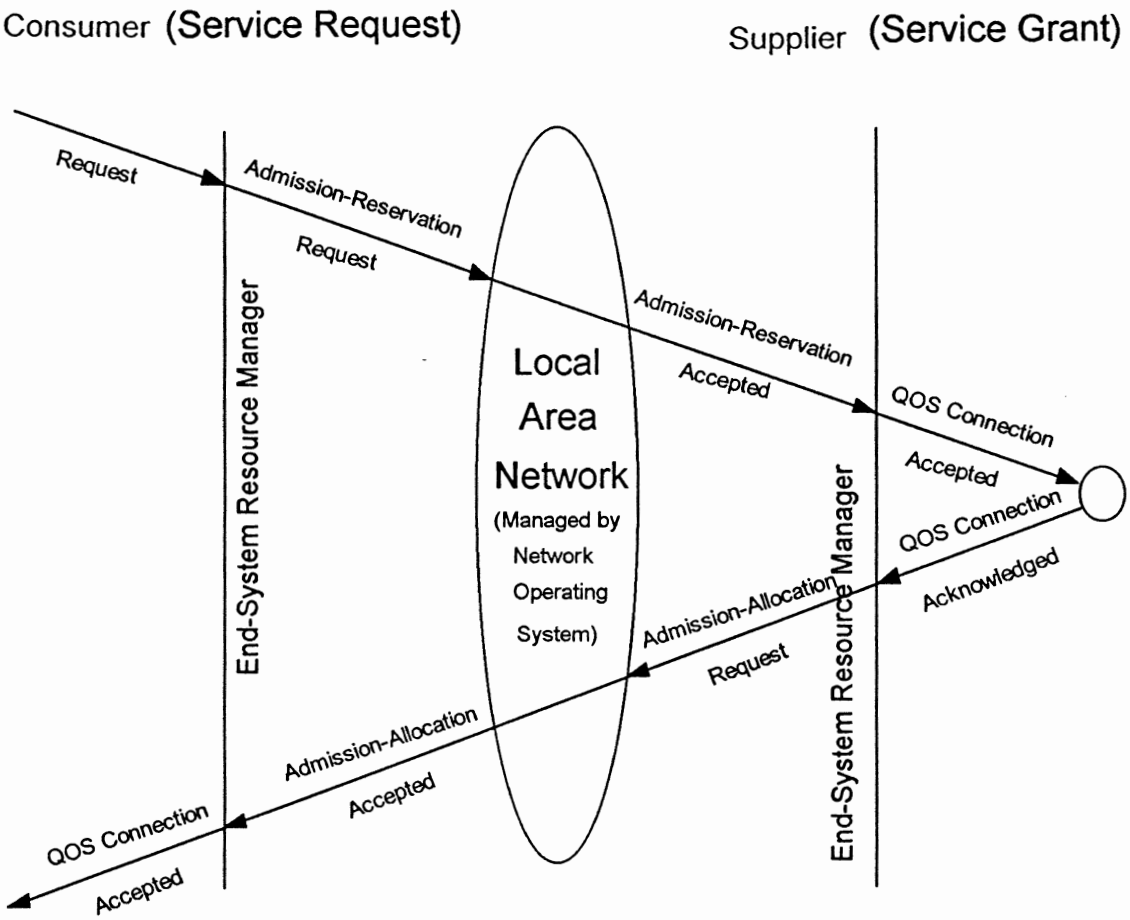

Figure 4.4: Resource reservation and resource allocation protocol for negotiating QOS requirements with an "accept" response (from [42]).

The supplier and the consumer both employ the layered QOS architecture shown in Figure 4.2. The application subsystem on either side orchestrates resources in the user space, which include memory buffers and processor utilization. On the other hand, the transport subsystem manages resources shared by lower layers of the network protocol stacks. The QOS profile registry stores and provides information necessary for resource synchronization. Global orchestration is achieved through interactions between the transport portion of the QOS negotiation agent and the network resource manager.

4.2.1 The Negotiation Protocol

An agreement on QOS requirements may not be reached during the negotiation for many reasons. The QOS negotiation agent may find that local resources are inadequate for the multimedia services requested; e.g., lack of memory buffers. Network resources may be insufficient to create a proper virtual circuit with enough bandwidth to accommodate the required media streams. Even if sufficient local resources are reserved, the remote supplier may be too busy serving other users and decide to drop the connection. In case the consumer QOS agent discovers that the requested QOS is denied by any of the mentioned entities, it sends a "*modify*" signal back to the application subsystem and attempts to lower the QOS requirements. The "*modify*" signal indicates the need for negotiation or renegotiation on the QOS parameters. Using the preferred,

acceptable, and unacceptable QOS values defined in Chapter 4, the QOS negotiation agent will reduce the QOS requirements from a preferred value (upper bound) to an acceptable value or down to just above the unacceptable value (lower bound). The range between preferred values and just above unacceptable values is referred as *"soft"* QOS guarantees. Soft guarantees mean that the QOS may change during the course of connection and renegotiation is needed to determine at what level the QOS will continue to be delivered.

The negotiation process is performed across the boundaries of all the layers within the application and transport subsystems as shown in Figure 4.5. The QOS negotiation agent incorporates several types of communication during negotiation: layer-to-layer, layer-to-operating system, peer-to-peer, peer-to-group, and group-to-peer. It is worthwhile to note that in ISO terminology, peer-to-peer negotiation is also known as *caller-to-callee negotiation* and layer-to-layer negotiation is called *service-user-to-service-provider negotiation*.

Layer-to-layer communication is used to facilitate the human user and application interactions. Layer-to-operating system communication is used during the admission of local system resources. Peer-to-peer communication is to obtain and distribute QOS resource requirements between the consumer and the remote supplier. Figure 4.6 provides a detailed description of the different types

of communication taking place across the layer boundaries of the proposed QOS management model.

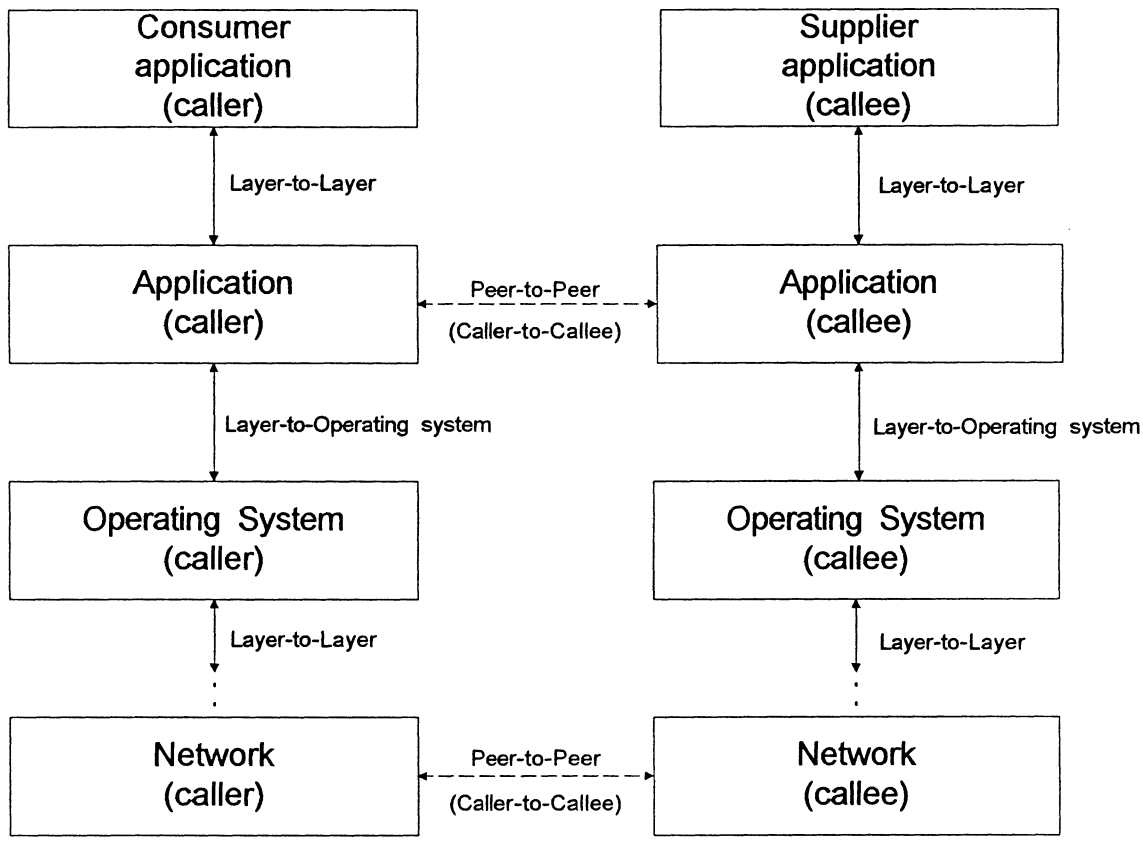

Figure 4.5: Negotiations across the layer boundaries of the proposed QOS management model.

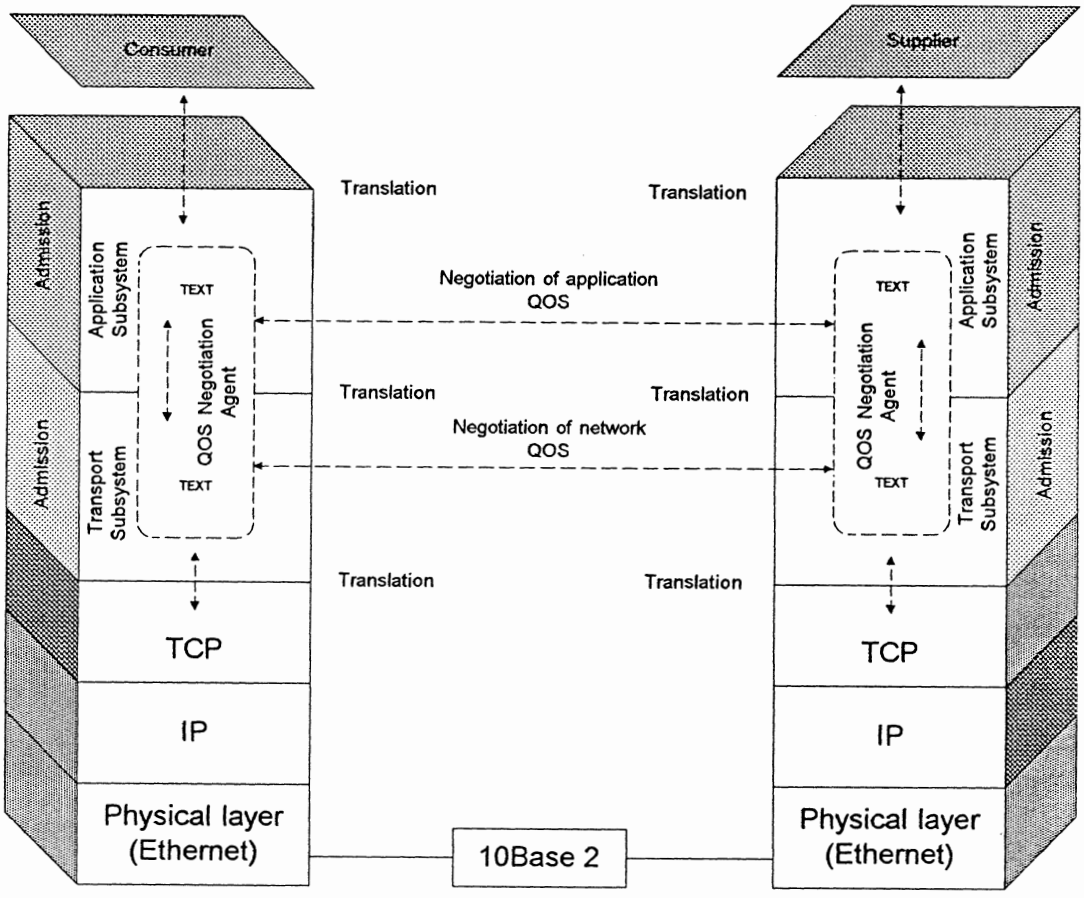

Figure 4.6: A detailed view of QOS negotiations.

During the negotiation process, signaling among the different layers in the application and network subsystems results in one of the three responses (see Figure 4.7):

i) *Accept*: The supplier agrees to allocate the resources at the remote site.

ii) *Modify*: The supplier cannot provide the required resources to the consumer at the preferred QOS levels. However, by relaxing the requested QOS specifications the supplier can still provide end-to-end services within the consumer's lower QOS bound.

iii) *Reject*: This signals that the supplier cannot provide the necessary resources even if the consumer reduces its QOS specifications to the lowest acceptable level. This may also be an indication of time-out problems or some irrecoverable errors taking place in the network.

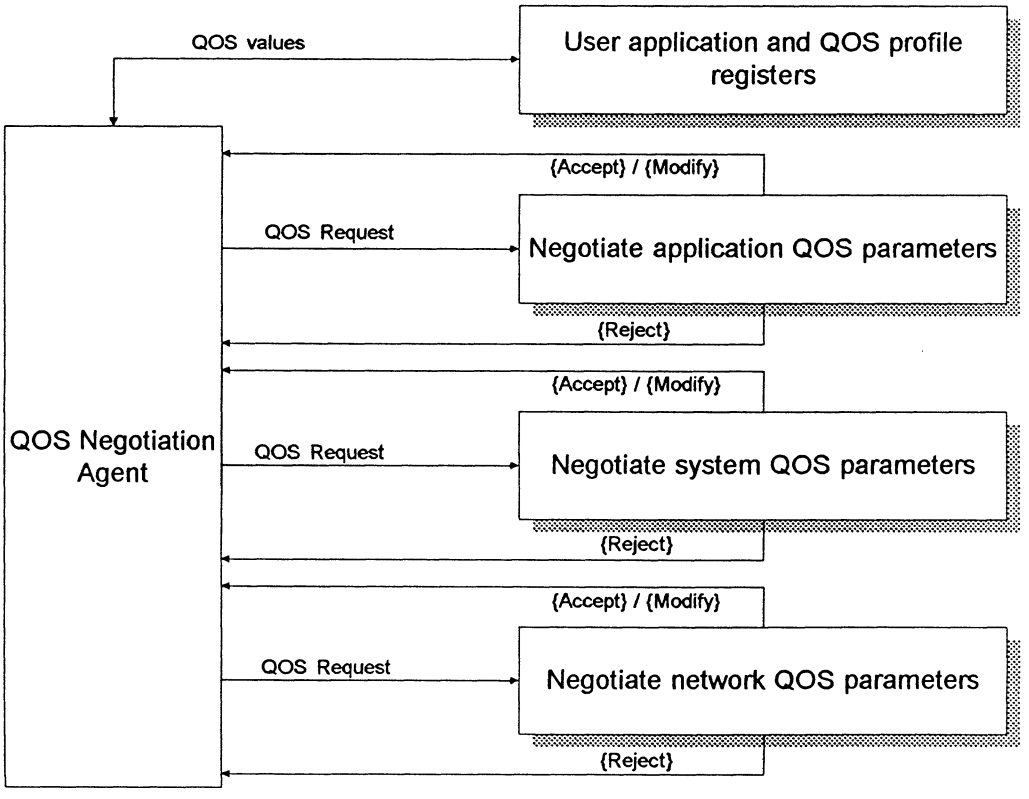

Figure 4.7: Signaling during QOS negotiations (from [43]).

To efficiently perform negotiations across the layers and among the peers, we have adopted several negotiation protocols:

i) Bilateral peer-to-peer negotiation is used at the application layer between peers as shown in Figure 4.8. This type of negotiation takes place between the consumer application subsystem and the supplier application subsystem. The consumer application specifies the QOS requirements and the supplier application is not permitted to modify the proposed value. The supplier QOS agent can, however, signal the consumer QOS agent that there is a need to modify the request QOS in order to conduct a successful session. Any modifications to the QOS parameters must be made by the consumer application.

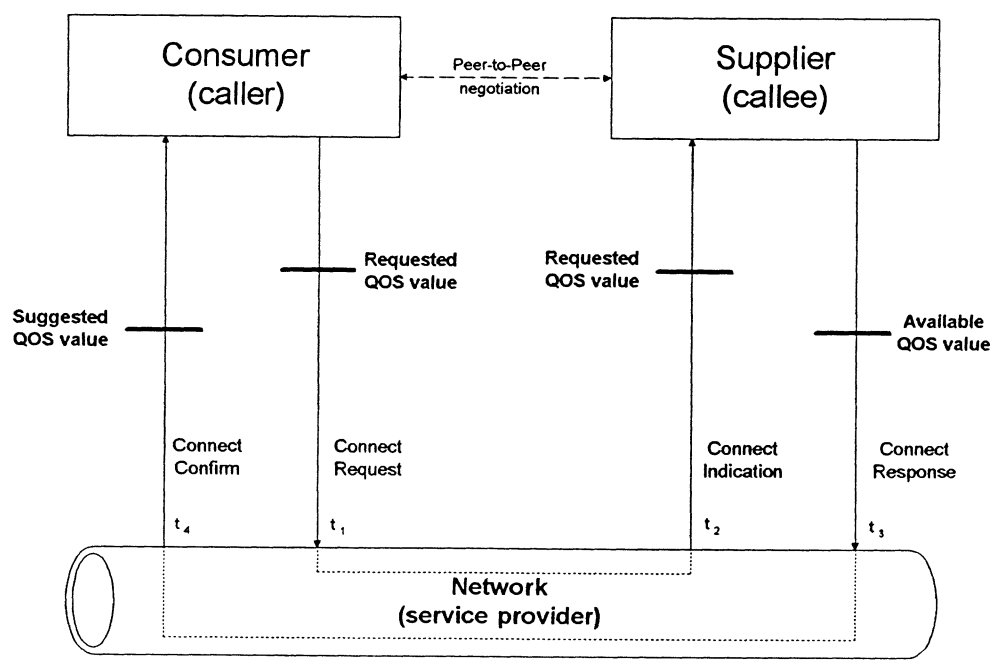

Figure 4.8: Bilateral peer-to-peer negotiation at the application layer between the consumer and the supplier.

ii) Unilateral layer-to-operating system negotiation is conducted between the application layer and the local operating system when local resources are being allocated. The operating system is not allowed to change the proposed QOS. However, the consumer application is allowed to control the presentation quality of the received media objects if necessary. For example, a digital audio data stream with stereo channels are multicasted to several users and one of the user machines can only play mono sounds. The machine that lacks stereo sound support will still be able to present the audio, but in monaural manner.

iii) Triangular negotiation is used at the transport subsystem layer to negotiate with the underlying network. Two methods of triangular negotiation may take place: triangular negotiation for a bounded target and triangular negotiation for a contractual value.

To allocate network resources, the QOS agent must negotiate at the transport layer. To begin the negotiation, the application QOS parameters are translated into the network QOS requirements. Triangular negotiation for bounded target is used to obtain the best possible QOS from the network. In this method of negotiation, the consumer QOS agent only presents the values of a QOS parameter through two bounds: the

preferred value (upper bound) and the acceptable value (lower bound). The objective is to negotiate for the preferred QOS value, which is the targeted value for this negotiation. The network manager is not permitted to change the lower bound that is set at the acceptable QOS value. However, it is allowed to modify the target value if it has determined that the target is too high to satisfy. The supplier QOS agent makes the final decision on whether target value suggested by the network is acceptable (see Figure 4.9).

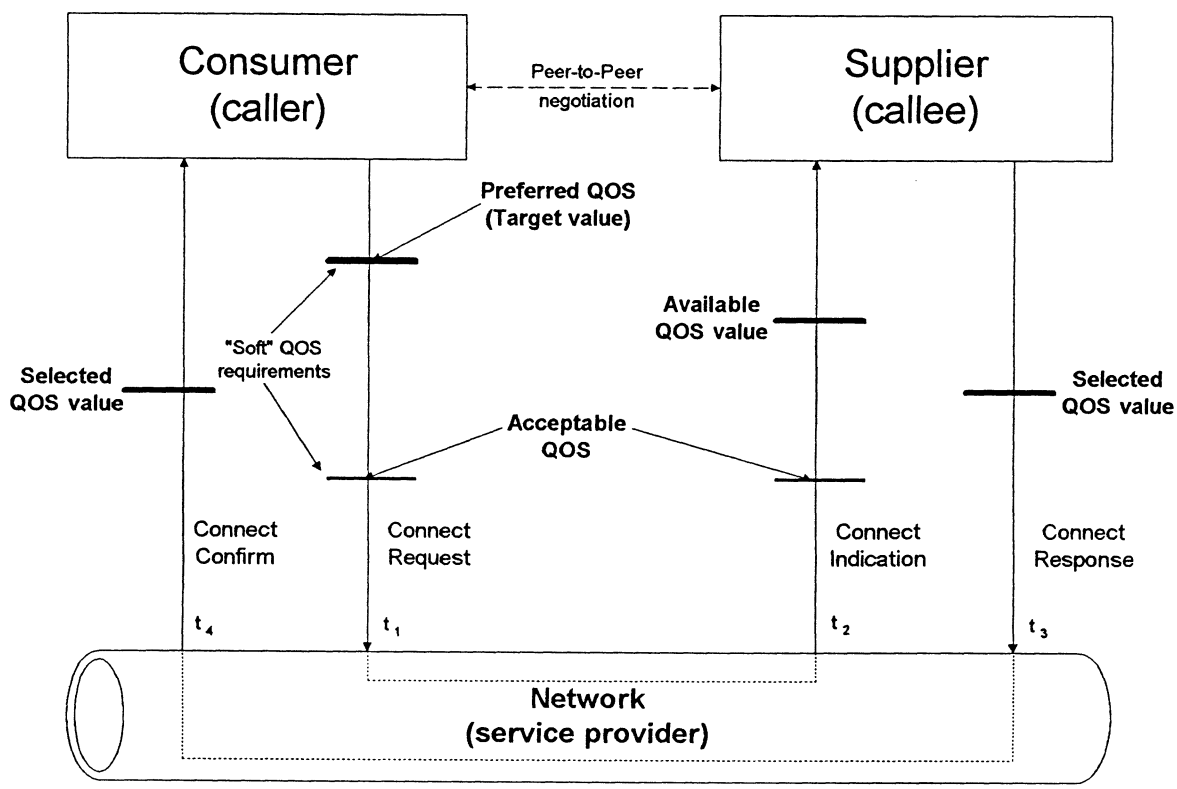

Figure 4.9: Triangular negotiation for bounded target.

If the supplier QOS agent agrees to accept the new target value, admission for the network resources will then be performed by the transport subsystem. If the supplier QOS agent cannot provide at least the acceptable quality, the second triangular negotiation is employed.

Triangular negotiation for a contracted value is used if the bounded target method fails. This happens when the supplier QOS agent cannot provide services at the acceptable QOS levels. In this case, the consumer QOS agent resubmits the QOS parameters using only the preferred and the unacceptable values. The objective here is to agree on a contractual value, which is set slightly above the unacceptable value, for each QOS parameter. The network resource management can increase the contractual value from the unacceptable level towards the preferred level as network resources permit. The supplier QOS agent makes the final decision and signals the consumer QOS agent. If the network cannot provide services at above the unacceptable QOS levels, the connection request is rejected without further negotiations. This situation is illustrated in Figure 4.10.

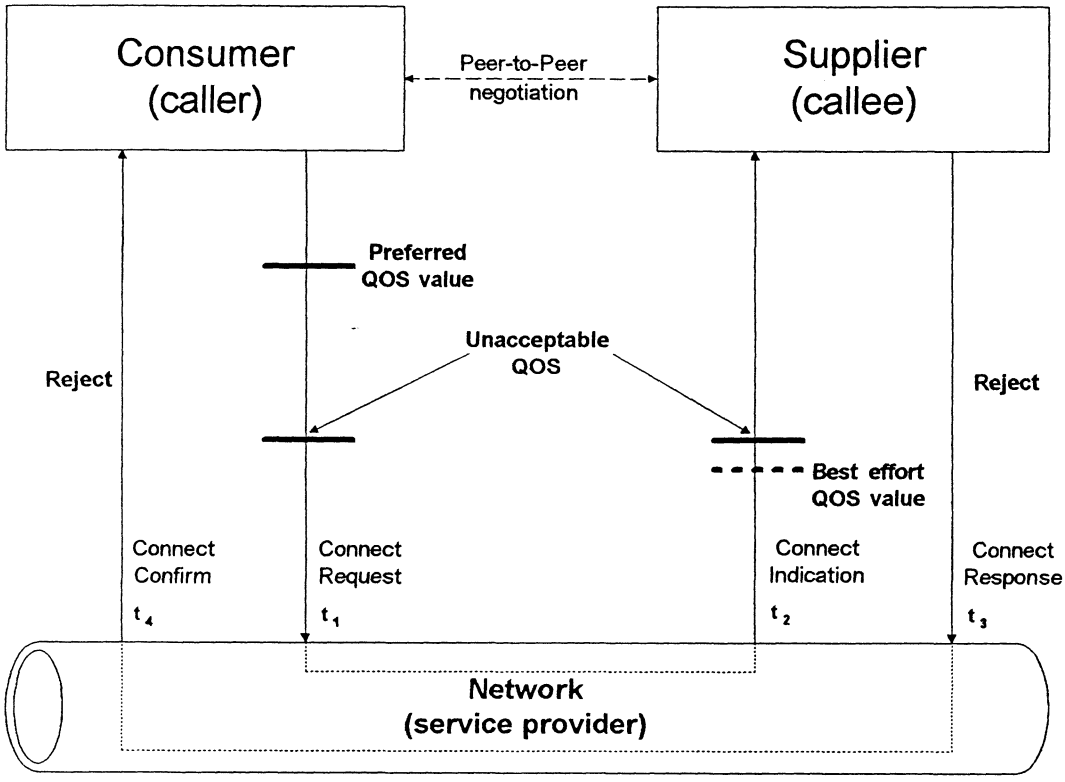

Figure 4.10: Triangular negotiation for contractual value.

During the course of a session, the user may reduce the size of a video window from 640 pixels by 480 pixels to 320 pixels by 200 pixels in order to get more viewing area for another application that is running concurrently. By reducing the screen size, the user may receive better service in terms of the frame rate per second since a smaller display window requires less processing power

and fewer memory buffers. This is an example of dynamic changes in QOS requirements during a multimedia session. Negotiations and modifications are continued to be made across the layers of the application and transport subsystem as the QOS requirements change. A separate virtual channel is used for exchanging QOS information between the consumer and the supplier without affecting the continuous media traffic (see Figure 4.11).

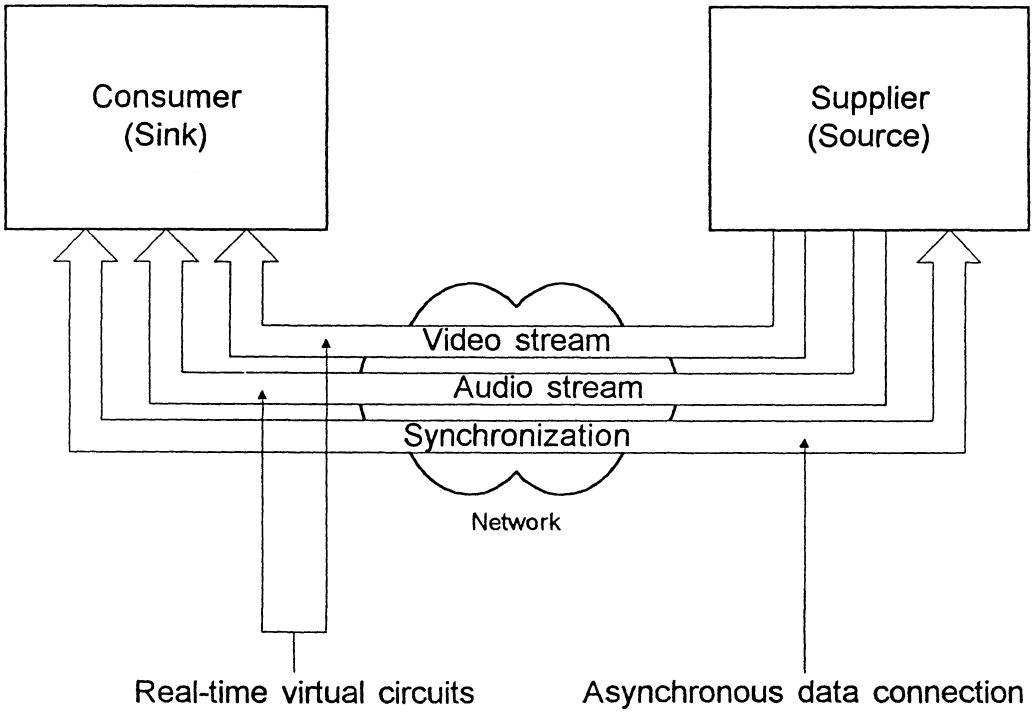

Figure 4.11: Using a separate channel to control synchronization.

4.2.2 The Consumer Protocol

The consumer protocol, as shown in Figure 4.12, is initiated by the input of application QOS requirements set at the user end-station. For instance, the user

may open a 640 pixels by 480 pixels window for displaying a video clip digitized at 256 colors. All the associated QOS parameters are stored in the application QOS profile registry.

The application QOS requirements are accessed from the QOS profile registry and then mapped into resource requirements for the local operating system. The consumer QOS agent negotiates with the operating system utilizing an admission service implemented in the QOS orchestration layer. The admission service assumes that task processing times and memory buffer space requirements are known a priori. This information must be available from the system QOS parameter profile registry before any admission decisions can be made. Two tests against the temporal resources are than performed by the admission service at the application level. A local schedulability test decides whether or not the tasks can manage I/O streams from multimedia devices within the required time bounds. An end-to-end delay test is also performed to determine whether or not the tasks can meet the specified end-to-end delay upper bound. The local resources are reserved if both tests are satisfied. At this point, the consumer QOS agent starts a peer-to-peer negotiation at the application level with the remote supplier QOS agent.

Since there is no reason to hold up shared network resources before we can admit the required local resources, the negotiation for application QOS is separated from the negotiation for network QOS. Admission of the local resources must be made before the negotiation with the network begins.

Unless the negotiation at the application layer is rejected, the consumer QOS agent initiates the request for network QOS requirements and begins network resource reservation and allocation. Four steps are carried out by the consumer QOS agent in the transport subsystem:

i) Application QOS parameters are translated into network QOS requirements.

ii) The admission service for the transport subsystem is initiated.

iii) Negotiation begins for per-connection network QOS parameters.

iv) Finally the consumer QOS agent waits for the network resource manager and the supplier QOS transport layer to reply. The accepted QOS values are translated back into application QOS and the QOS profile registry is updated.

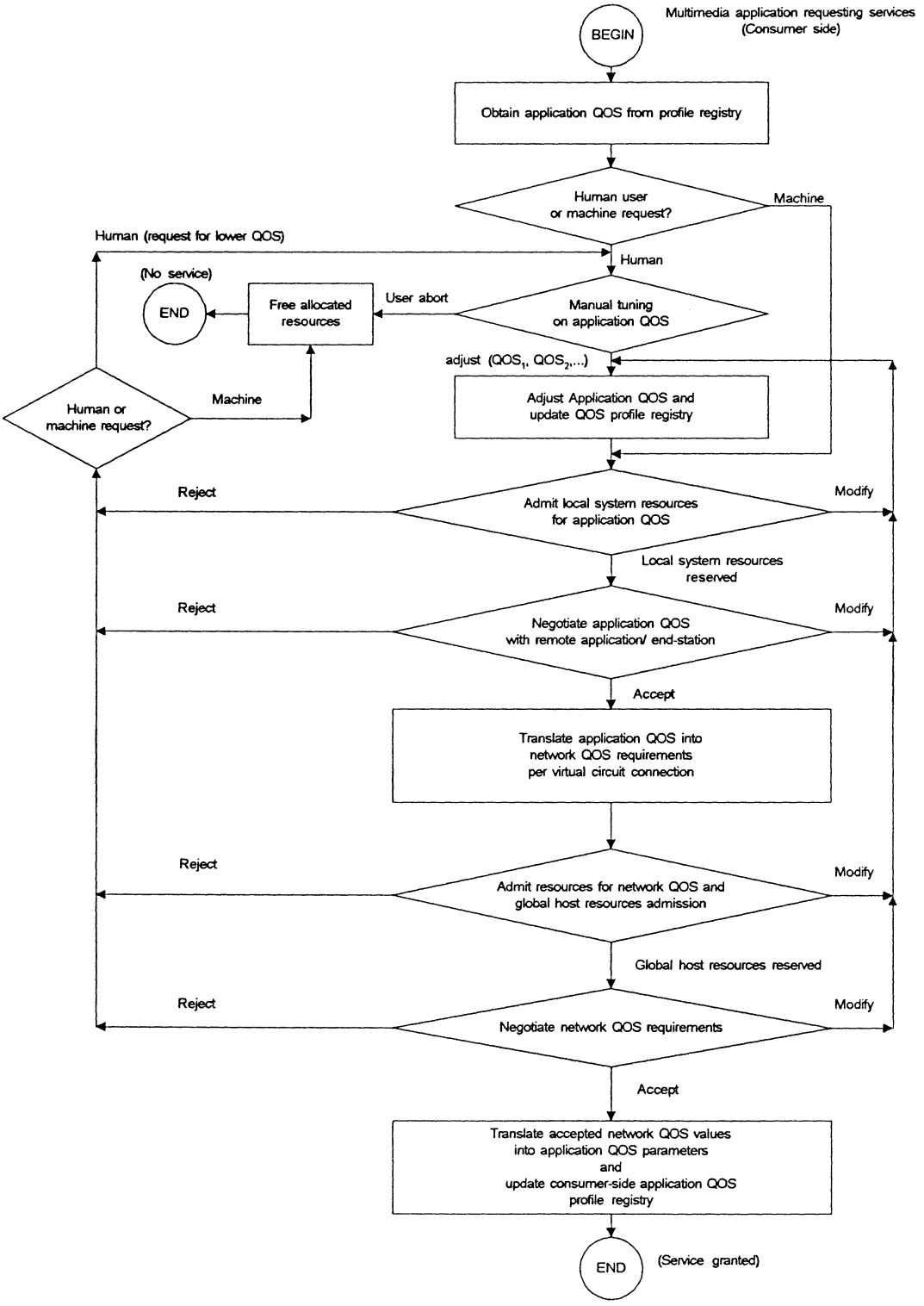

Figure 4.12: Flowchart for the consumer protocol.

4.2.3 The Supplier Protocol

The supplier protocol works similarly to the consumer protocol as shown in Figure 4.13. The supplier QOS agent responds to the consumer call for service by waiting for the remote consumer to send the requested application QOS parameters. The received application QOS parameters are compared against and the supplier's own application QOS output parameters. A match between the two sets of QOS values invokes the admission service in the supplier application subsystem. The supplier QOS agent signals the remote consumer according to results from the admission service. A positive negotiation of the application QOS parameters is followed by obtaining the consumer's network resource information from the network resource management layer. The network resource management then signals the global admission service at the supplier transport subsystem. The negotiation protocol translates the network QOS parameters into the application QOS requirements and determines if resources can be allocated, relaxed or released according to the availability of resources. If all required resources are available, they are allocated and the multimedia session begins.

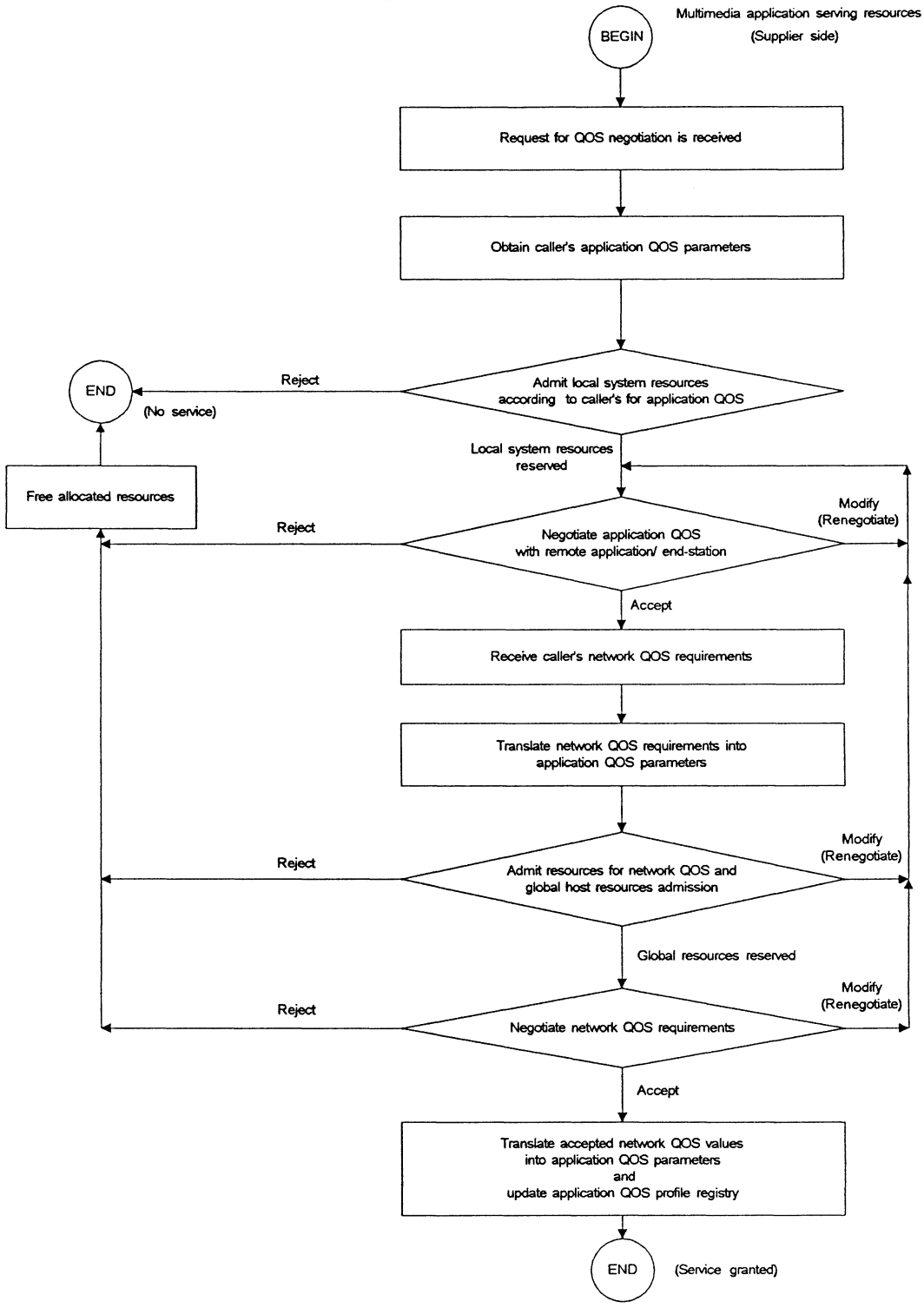

Figure 4.13: Flowchart for the supplier protocol.

4.3 Group Communication

When multiple consumers or suppliers are involved in a multimedia session, peer-to-group and group-to-peer communication methods are utilized. Peer-to-group communication is needed when the consumer wants to receive multimedia streams from several remote suppliers. The consumer application specifies the addresses of each remote supplier and the types of services requested from the identified suppliers. At the transport level of the consumer side, the network management system provides the network addresses and waits for responses returned from the network and the group of suppliers. At this point the QOS negotiation agent on the consumer side must multicast the network QOS requirements to all members involved in this group communication session. The agent relies totally on the multicast capabilities of the underlying network. Neither the agents nor the end-point transmission protocols have multicasting capabilities.

The suppliers at various locations proceed resource allocation according to the consumer QOS agent specifications and return their resource management decisions to the consumer QOS agent. If the suppliers cannot deliver the service at the specified quality, they must modify their own capabilities at the transport subsystem. Each supplier is responsible for adjusting its own QOS capabilities. The advantage of negotiating with each supplier separately is that each supplier

can report its own offerings to the consumer. The disadvantage is that the number of connections required to conduct the negotiations increases proportionally with the total number of group members involved. Figure 4.14 shows the negotiation paths in peer-to-group communication.

Group-to-peer communication is essentially identical to the peer-to-group communication except that the consumer must decide how to allocate resources for multiple incoming connections. The remote suppliers may return different QOS specifications and it is the duty of the consumer to deal with resource allocation and management for all the incoming media streams. The distribution of negotiation messages is also similar to that of the peer-to-group communication.

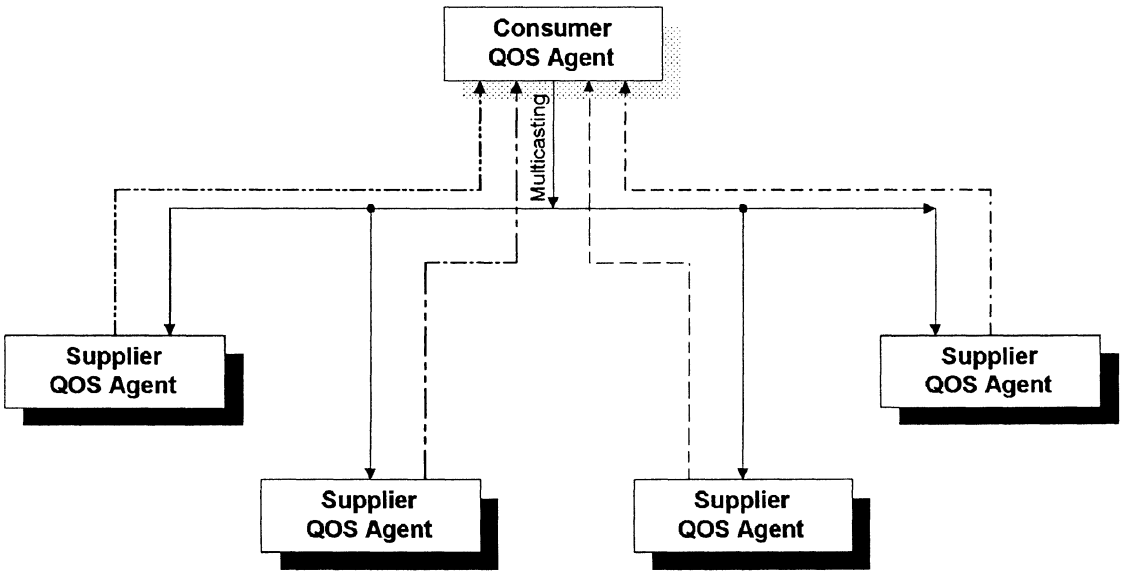

Figure 4.14: The negotiation paths in group communication.

In this Chapter, we have proposed on the architecture of our integrated QOS management model. Using a layered architecture across the application and the transport subsystems, an application can negotiate and establish an end-to-end multimedia communication efficiently. Based on a supplier-consumer model, we have developed the QOS negotiation agents that conduct QOS negotiations on behalf of a consumer (e.g., a human user) and a supplier (e.g., a video server) of multimedia services. We have also discussed the negotiation protocol, the consumer protocol, and the supplier protocol employed by the QOS negotiation agents. In the next Chapter, we describe the computer implementation of this model and present the experimental results.

Chapter 5

Computer Implementation and Experimental Results

The integrated QOS management architecture proposed in this research is motivated by the stringent timing requirements imposed by distributed multimedia applications and the lack of continuous media management in the current commercial operating systems. To implement the QOS negotiation model, we need real-time support from the operating system (OS). Single-tasking OS such as MS-DOS cannot support continuous media in a distributed environment. For this reason, we have selected Microsoft Windows NT v.3.51 as the platform for implementing and testing our simulation programs mainly because it provides real-time multi-tasking support. Another important factor for this selection is that Windows NT includes the Windows Socket library, called WinSock [44], which provides many networking functions for implementing TCP/IP (Transmission Control Protocol/Internet Protocol) compliant applications. A discussion on WinSock is included in Appendix A.

5.1 Priority Inversion Problem

When a real-time application shares the same resources in a system with non real-time applications, the real-time application is usually forced to wait for the completion of the non real-time applications. For instance, if many non real-

time applications, such as file utilities, word-processors or spreadsheets, are sharing the same network file server, the data packets of a high-priority task's video stream must wait for the completion of all previously queued low-priority packets. If the operating system does not preempt the low-priority tasks and allow the high-priority tasks to execute first, unexpected delay and jitters will result. This problem is called priority inversion.

Priority inversion is avoided by exploiting the strength of preemptive multitasking in Windows NT. Windows NT uses a set of priority queues to determine a thread's eligibility to execute when it is time for context switching. The Windows NT dispatcher examines the set of ready threads and selects the head of the highest priority queue to be executed first. For example, the thread for handling time-critical video packets, that has a higher priority, will be executed immediately by preempting the currently running non real-time thread, that has a lower priority. When it is the time for the preempted thread to use the processor again, the operating system restores the state of the thread and allows it to resume execution. The Windows NT process object is depicted in Figure 5.1, showing the process address space and several different threads.

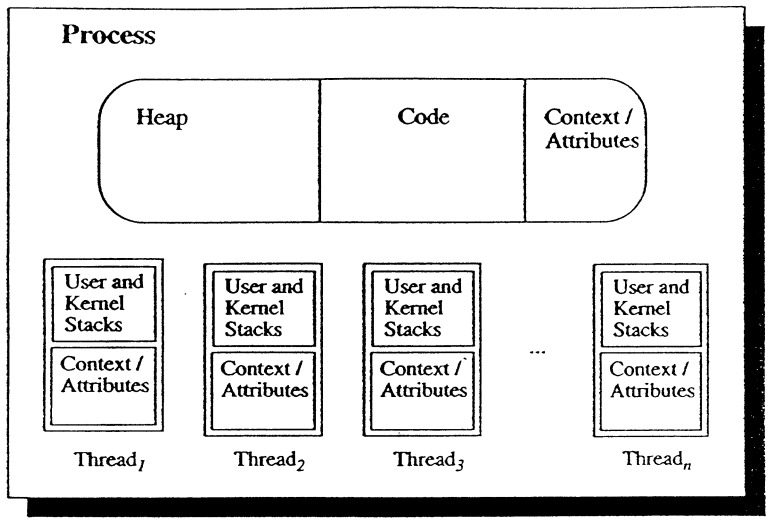

Figure 5.1: Process and threads in Windows NT (from [45]).

The priority of a thread is determined by the thread's process priority class, and its base and dynamic priorities. Within each priority level, threads are scheduled using a first-come-first-served round-robin policy. Figure 5.2 shows the possible states of a thread in Windows NT.

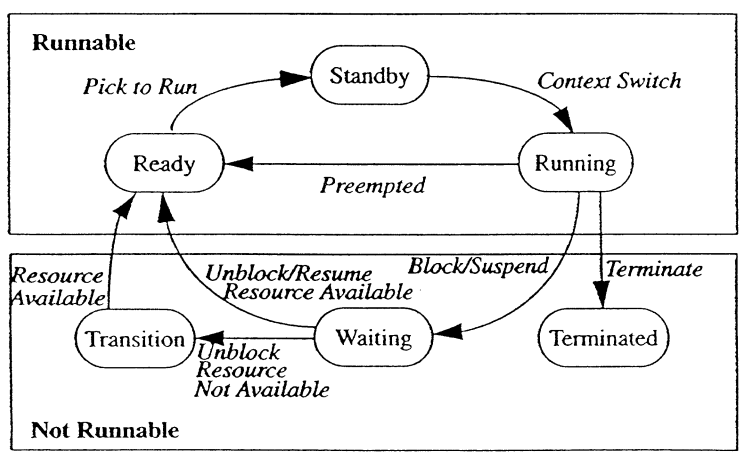

Figure 5.2: Possible states of a Windows NT thread (from [45]).

The thread's process priority class is the most important component of a thread's priority. In Windows NT, thread priority can be one of the four classes: *real-time, high, normal,* and *idle.* By default, a process is started with the NORMAL_PRIORTY_CLASS. When QOS negotiation begins, the processes involved in the negotiation are set with the HIGH_PRIORITY_CLASS. Once the admission of resources is completed, multimedia streams begin to flow from the source to the user.

To determine the priority class of a process, the GetPriorityClass function is used:

```
1: DWORD PriorityClass;
2: HANDLE hProcess;
3: hProcess = GetCurrentProcess();
4: PriorityClass = GetPriorityClass( hProcess );
```

GetPriorityClass uses a process handle (hProcess) as the only argument and returns the priority class of that process or the zero value in case of an error.

To modify the priority class of a process, the SetPriorityClass function is used. SetPriorityClass uses two argument: a process handle and the new process priority class. An example follows:

```
1: DWORD error_code;
2: BOOL State;
3: HANDLE hProcess;
4: hProcess = GetCurrentProcess();
5: State = SetPriorityClass(hProcess, HIGH_PRIORITY_CLASS);
6: If (State == FALSE)
7:    error_code = GetLastError();
```

This function should be used with caution since raising the priority class of a process to HIGH_PRIORITY_CLASS may starve all other processes in a lower priority class.

Within each priority class, there is a thread base priority level. There are a total of five thread priority levels:

1. THREAD_PRIORITY_HIGHEST
2. THREAD_PRIORITY_ABOVE_NORMAL
3. THREAD_PRIORITY_NORMAL (the default level)
4. THREAD_PRIORITY_BELOW_NORMAL
5. THREAD_PRIORITY_LOWEST

All newly spawned threads have the priority of THREAD_PRIORITY_NORMAL by default. An application can determine a

thread's priority by calling the GetThreadPriority function. This function accepts a thread handle as the only argument and returns that thread's priority. If the function call returns the value THREAD_PRIORITY_ERROR_RETURN, this indicates that an error has occurred. An example is given below:

```
1: DWORD error_code;
2: BOOL State;
3: HANDLE hThread;
4: int ThreadPriority;
5:
6: hThread = GetCurrentThread();
7: ThreadPriority = GetThreadPriority(hThread);
8: If (ThreadPriority == THREAD_PRIORITY_ERROR_RETURN)
9:    error_code = GetLastError();
```

A thread's base priority can be raised or lowered by calling the SetThreadPriority function with two arguments: a thread handle and an integer priority level as mentioned above. The function returns a Boolean value indicating whether or not the operation is successful. By setting the priority above the normal level, a thread can request service from the operating system in order to handle some continuous media. The thread should not stay running at high

priority after handling time-critical data to avoid *starving* other threads that are running at a lower priority. An example is given below:

```
1: DWORD error_code;
2: BOOL State;
3: HANDLE hThread;
4: int ThreadPriority = THREAD_PRIORITY_ABOVE_NORMAL;
5: . . .
6: hThread = GetCurrentThread();
5: State = SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST);
6: If (State == FALSE)
7: {  error_code = GetLastError();
8:    // Error handling begins
9:    . . .
10: } // end_if
11: else
12: { //Execute some  job at high priority
13:   . . .
14:   // Return to normal Priority
15:   State = SetThreadPriority(hThread,THREAD_PRIORITY_NORMAL);
16:   If (State == FALSE)
17:   {     error_code = GetLastError();
18:         // Error handling begins
19:         . . .
20:   }  // end_if
```

```
21: } // end_else
```

In addition to a base priority level that is changeable by the thread itself, a thread also has a dynamic priority level that can be altered by Windows NT. The operating system employs this function when it needs to make a thread more responsive to certain events by raising the thread's priority level. The level of priority promotion depends on the type of event that the thread awaits. In Windows NT, the scheduling policy is that threads awaiting keyboard input receive the highest amount of priority promotion so that they can be responsive to user inputs. On the other hand, threads awaiting I/O events receive a medium amount of promotion and threads that are computer-bound get the least promotion. After the event has passed, the scheduler lowers the dynamic priority by one level at each time slice until the thread priority returns to its base priority. Hence, the operating system can never lower a thread's priority level beyond its original base priority. A summary of Windows NT thread interface calls is shown in Figure 5.3.

Windows NT Thread Interface Call	Description
CreateThread	Create a new thread
CreateRemoteThread	Create a new thread in a different process address space
GetCurrentThread	Return a pseudo handle to the current thread
SuspendThread	Suspend a specified thread's execution
ResumeThread	Resume the execution of the specific thread
ExitThread	Terminate the current thread
TerminateThread	Terminate a specified thread
GetThreadPriority	Get the base priority of the specified thread
SetThreadPriority	Set the base priority of the specified thread
DuplicateHandle	Get a duplicate handle to a thread object
CloseHandle	Relinquish a thread handle
WaitForSingleObject	Wait for the specified object to attain a signaled state
WaitForMultipleObjects	Wait for all or one of many specified objects to attain a signaled state
CreateEvent	Create an event synchronization object
SetEvent	Signal an event
ResetEvent	Set the state of an event object to not-signaled
PulseEvent	Set and then reset an event
InitializeCriticalSection	Initialize a critical section object
EnterCriticalSection	Acquire a critical section object
LeaveCriticalSection	Release a critical section object
DeleteCriticalSection	Remove a critical section object from the system
CreateMutex	Create a mutex synchronization object
ReleaseMutex	Signal a mutex object
OpenMutex	Given the mutex name, obtain a handle to it
CreateSemaphore	Create a new semaphore synchronization object
ReleaseSemaphore	Signal a semaphore object
TlsAlloc	Allocate the thread local storage
TlsSetValue	Set the value of a thread local storage
TlsGetValue	Get the value of a thread local storage
TlsFree	Free or de-allocate the thread local storage

Figure 5.3: Summary of Windows NT thread interface calls.

5.2 QOS-Based Resource Control

The dynamic QOS control scheme that we use with the QOS negotiation agent allows the initial QOS values to change during the course of a multimedia session. The following pseudo code demonstrates how dynamic changes in QOS are handled in the program:

```
1: LONG APIENTRY MainWndProc(HWND hwnd, UINT message,
2:                           UINT wParam, LONG lParam)
3: {
4:    DWORD  ThreadID1, ThreadID2, ThreadID3;
5:    static HANDLE hThread1, hThread2, hThread3;
6:    . . .
7:    main_thread_body;
8:    . . .
9:    session_create (qos_manager, qos_request);
10:   . . .
11:   session_control (qos_manager, qos_change);
12:   . . .
13:   session_callback(session, qos_level)
14:   {     modify_qos(session, qos_level);
15:   }
16:   . . .
17: }
```

After creating a session using the `session_create` procedure in line 9, the user application may submit a request for degrading the initial QOS parameters; e.g., rescaling the video window to a smaller size. This is handled by the `session_control` procedure in line 11. Meanwhile, the QOS negotiate agent may invoke a call-back function (line 13) for restoring or degrading the QOS values of the session being processed when either the application subsystem or the transport subsystem signals the need for such a change in QOS levels.

5.3 QOS Management and Admission Control

In order to coordinate with the QOS negotiation agent in performing dynamic QOS control, admission services for QOS allocation must also work in accord. The pseudo code of the admission control for managing changes in QOS values is given below:

```
1:    qos_control ( )
2:    {     . . .
3:          accept_request ( );
4:          switch (sigal)
5:                case ADMISSION_TEST:
6:                      estimate_resource_request ( );
7:                      check_mem_buffers ( );
8:                      check_schedulability ( );
```

```
9:                    check_network_capacity ( );
10:                   if  (IS_REQUEST_ACCEPTABLE)
11:                    qos_level = determine_qos_init_value( );
12:                   replay (caller, qos_level);
13:                   . . .
14: }
```

This implementation allows the initial value of the requested QOS level to be admitted if all the resource allocation checks are passed. Windows NT also provides several synchronization mechanisms known as synchronization objects. They are events, critical sections, mutex (mutual exclusive), and semaphores. These mechanisms are widely used in operating system development [46].

5.4 Experimental Results

The implementation of the integrated QOS management model requires that the existing operating system's kernel to be modified in order to support the real-time operations of the proposed architecture. Since the source code for Windows NT is not commercially available, we can only write simulation programs to study the effectiveness of the QOS management scheme developed here. The QOS negotiation agent is written as a background process handling all the negotiation threads by monitoring the operating system resources, the network resources, and the state of the multimedia application that is running in the foreground. The multimedia application that we developed simulates an AVI

(Audio/Video Interleaved) digital video player. The AVI format is the native file format defined by the digital video framework from Microsoft called Video for Windows (VfW). Additional information on the AVI file format is given in Appendix B.

The AVI player can run in two modes: one with QOS negotiation capabilities enabled and the other with the QOS negotiation capabilities disabled. When it is running in QOS negotiation mode, it utilizes the QOS negotiation agent to adjust QOS levels according to available resources. With QOS negotiation mode disabled, the simulated AVI player relies only on the standard functions provided by Video for Windows which has no QOS management.

The experiments are conducted on a on a local area network with a single server and four workstations (see Figure 5.4). The server is running Windows NT Server 3.51, while the workstations are running Windows NT Workstation 3.51. The network runs TCP/IP on 10-Base2 Ethernet. The server and two of the workstations are Intel Pentium® computers each having 32MB of RAM and a clock frequency of 60MHz. They are configured similarly, each with a 16-bit sound card, 64-bit PCI local bus video adapter with 2MB video RAM, and a quad-speed CD-ROM. The two remaining workstations are Intel 486-DX2® machines running at 66MHz.

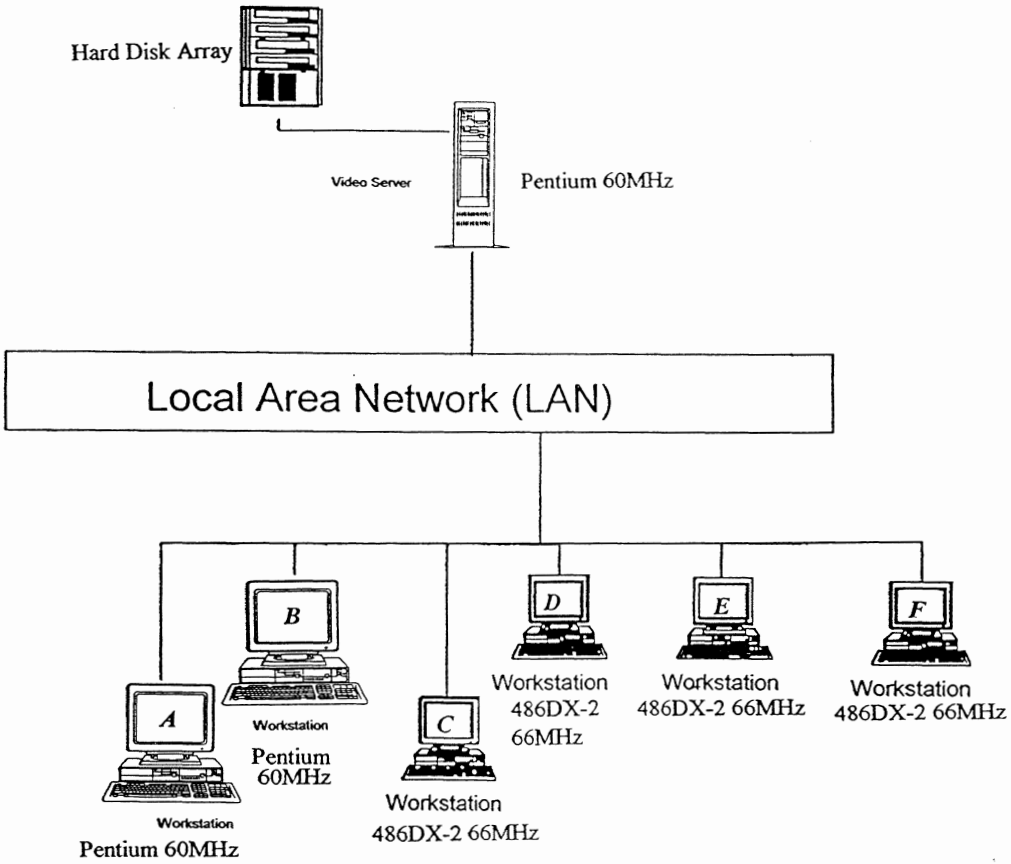

Figure 5.4: The LAN system setup for conducting the QOS management experiments.

Both are equipped similarly, each having 16MB RAM, a 16-bit sound card, 64-bit PCI local bus video adapter with 2MB video RAM, and a double-speed CD-ROM. The use of different machines in the computer simulation enables us to observe whether or not the heterogeneous nature of a LAN has any impact on the QOS each machine can deliver. In the remaining part of this Chapter, the two Pentium® workstations are referred as machines *A* and *B* and the two 486-DX2® workstations are referred as machines *C* and *D*.

The first experiment involves the measurement of the average number of frames per second under three conditions: One, two, and four video sessions are initiated by the AVI player running on each machine simultaneously, each reading a different copy of the same AVI file. The playback file is originally recorded at 160 pixels by 120 pixels at 25 fps without compression and no skew or jitters occurred during capture. Four copies of the same file are stored in the server under different file names. The files are read off from the server and played as the data streams get served through the network. The spatial resolution of the playback window for each machine is set to 320 pixels by 200 pixels (by doubling the original captured resolution during playback) with 8-bit color depth (256 colors). Figure 5.5 shows the results of the AVI players running with the QOS negotiation mode disabled. The results indicate that the player can achieve up to 23 fps (using machine *A*) when only a single session is running. There are

little fluctuations in the frame rate and the rate is stabilized throughout the 15-second playout time. As more sessions run simultaneously, the frame rate drops to 11 fps for two sessions (using machines *A* and *B*) and down to an unbearable 5 fps for four sessions. Variations in the frame rate are more severe when four sessions are running.

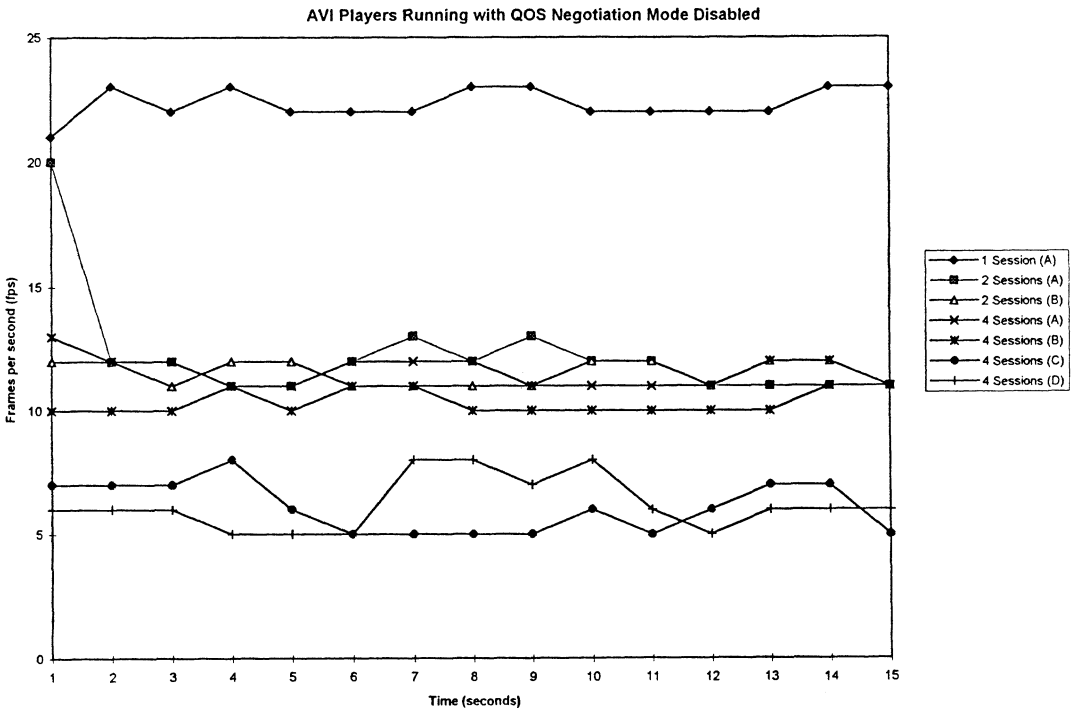

Figure 5.5: AVI players running with the QOS negotiation mode disabled. Separate plots of the results are available for: One session (Figure 5.6), two sessions (Figure 5.7), and four sessions (Figure 5.8).

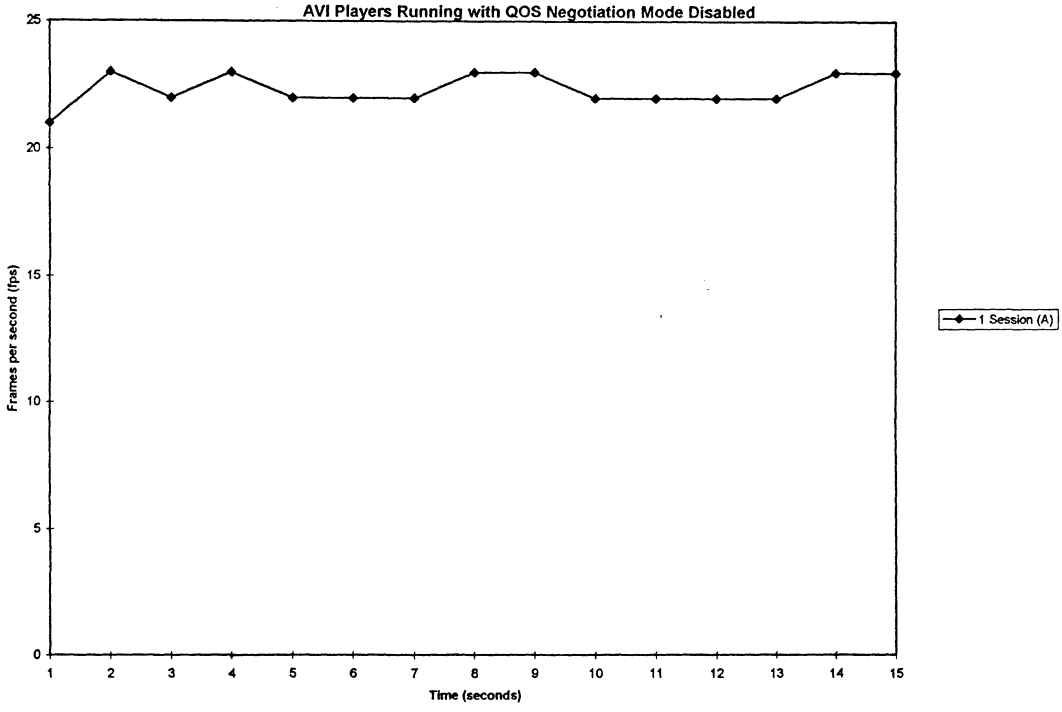

Figure 5.6: A single AVI player running with the QOS negotiation mode disabled.

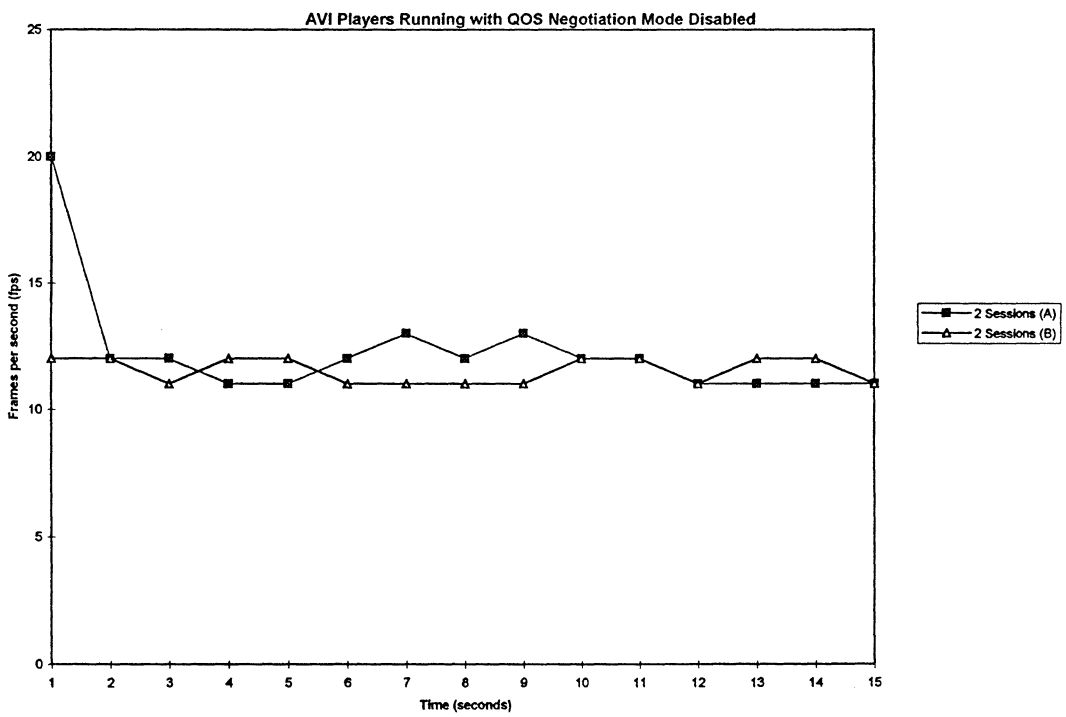

Figure 5.7: Two sessions of the AVI players running with the QOS negotiation mode disabled.

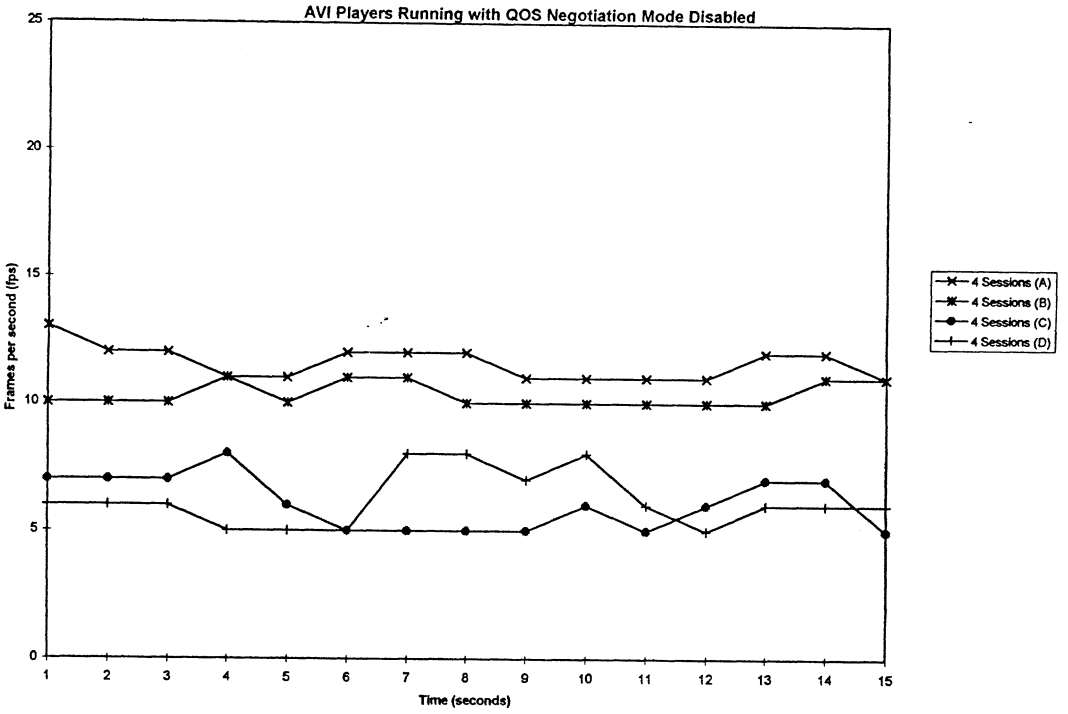

Figure 5.8: Four sessions of the AVI players running with the QOS negotiation mode disabled.

To determine the jittering and skew effects between frames, we have measured the interframe gap and the results are shown in Figure 5.9. The results indicate that interframe gap varies greatly when four sessions are running simultaneously. This happens when frames are dropped, attempting to correct the effects of jitter and skew.

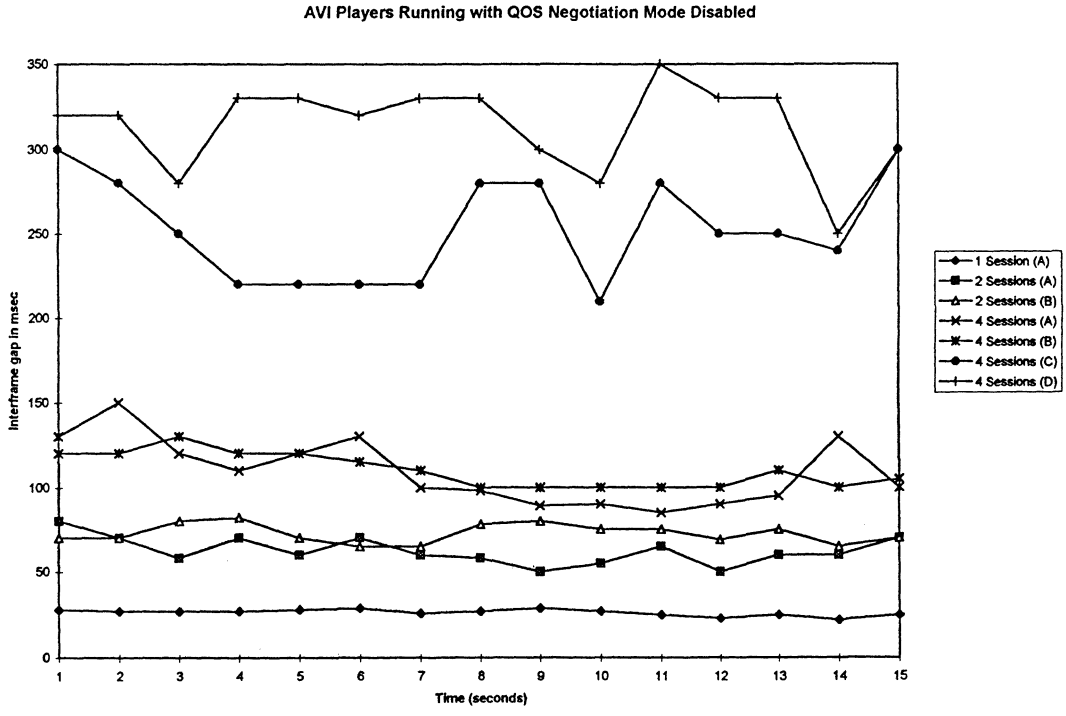

Figure 5.9: Interframe gaps measured by running the AVI players with the QOS negotiation mode disabled. Separate plots of the results are available for: One session (Figure 5.10), two sessions (Figure 5.11), and four sessions (Figure 5.12).

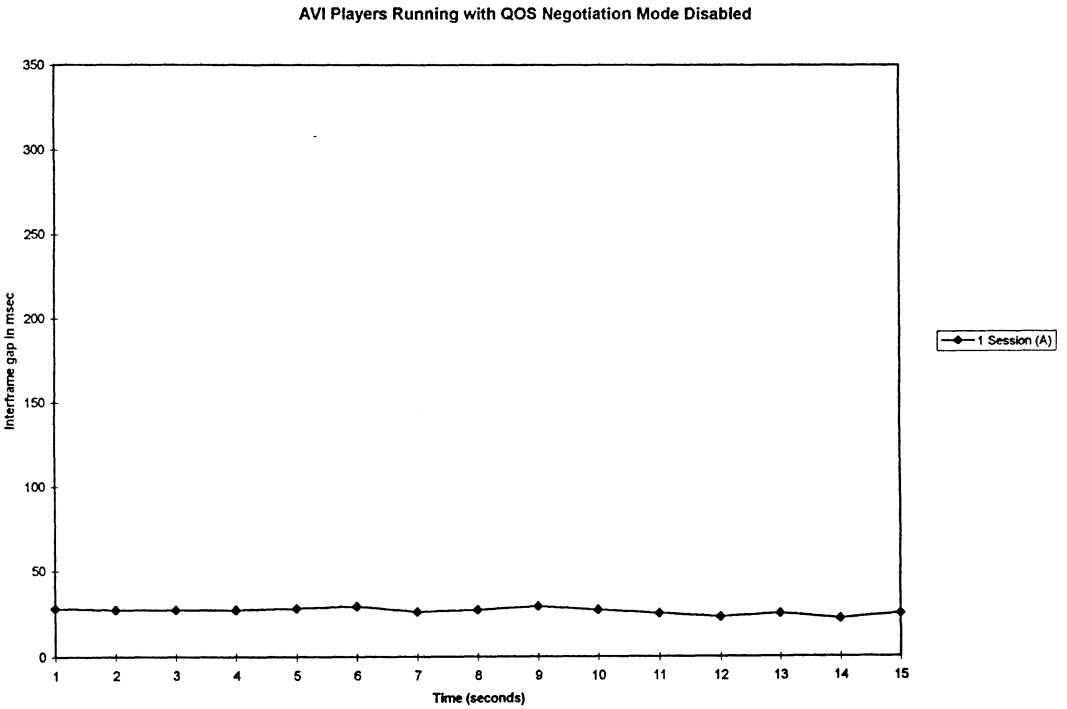

Figure 5.10: Interframe gaps measured by running a single session of the AVI player with the QOS negotiation mode disabled.

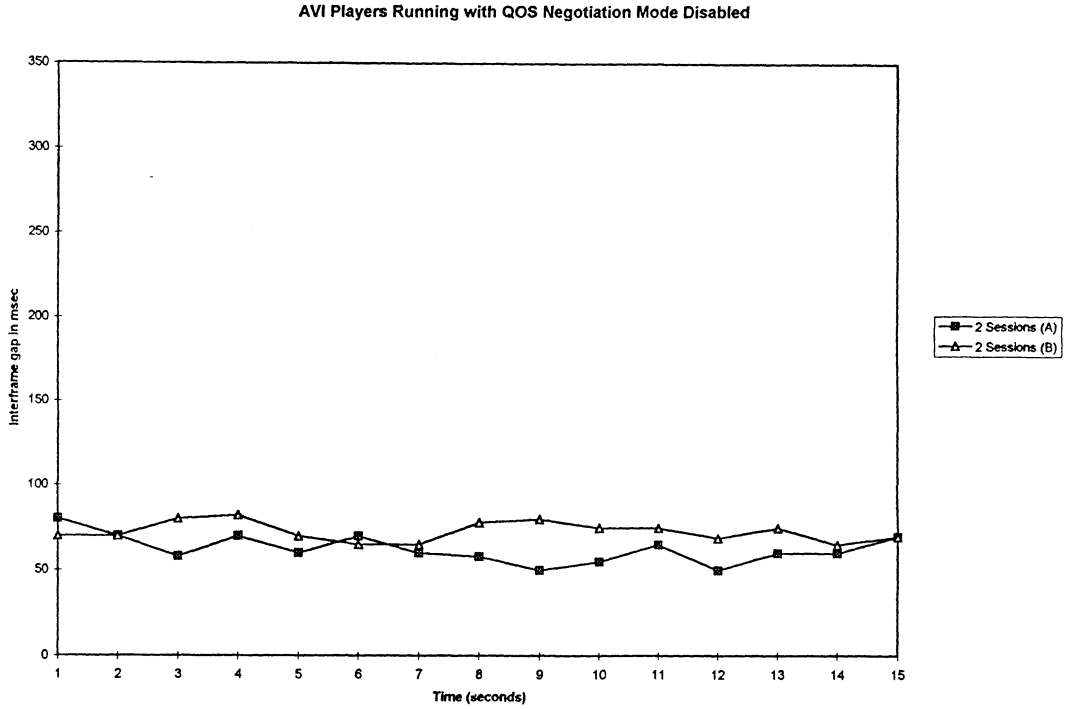

Figure 5.11: Interframe gaps measured by running two sessions of the AVI players with the QOS negotiation mode disabled.

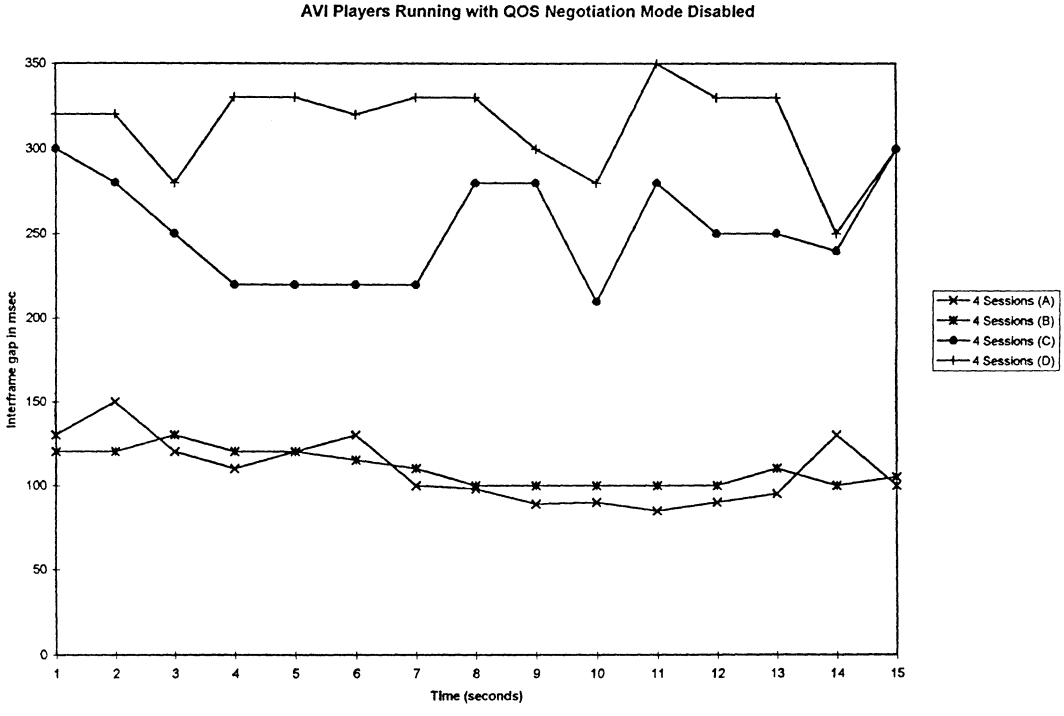

Figure 5.12: Interframe gaps measured by running four sessions of the AVI players with the QOS negotiation mode disabled.

Next, we run the experiment again with the same conditions as before, except that the QOS negotiation mode is enabled. For the purpose of comparing results, each session keeps the original resolution and color depth so that none of the sessions can reduce these parameters in order to compensate for the degraded QOS. The results shown in Figure 5.13 indicate that the player still achieves up to 23 fps when only a single session is running (machine *A*). In this case, fluctuations in the frame rate are limited and the rate is stabilized throughout the 15-second playout time. The frame rate drops to 12 fps with two sessions running but the frame rate stabilizes at 12 fps for the duration of the test. When four session are running simultaneously, the session running on machine *D* goes down to 5 fps and is aborted after three seconds of playout time have elapsed. This is because the unacceptable QOS level is set to 6 fps and machine *D* sees no hope to continue the session at an acceptable QOS level. The remaining sessions continue to run at approximately 10 fps for the rest of the test. Interframe gaps are also small and stay at stable levels after machine *D* terminates its session and leaving only three sessions running (see Figure 5.17).

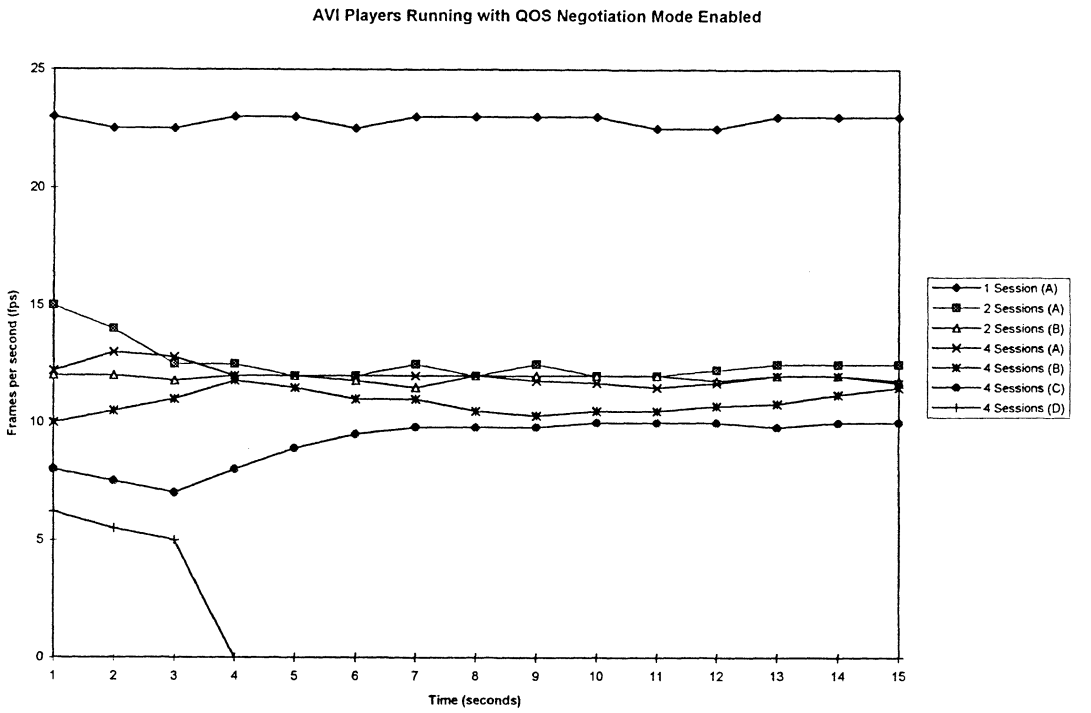

Figure 5.13: AVI players running with the QOS negotiation mode enabled. Separate plots of the results are available for: One session (Figure 5.14), two sessions (Figure 5.15), and four sessions (Figure 5.16).

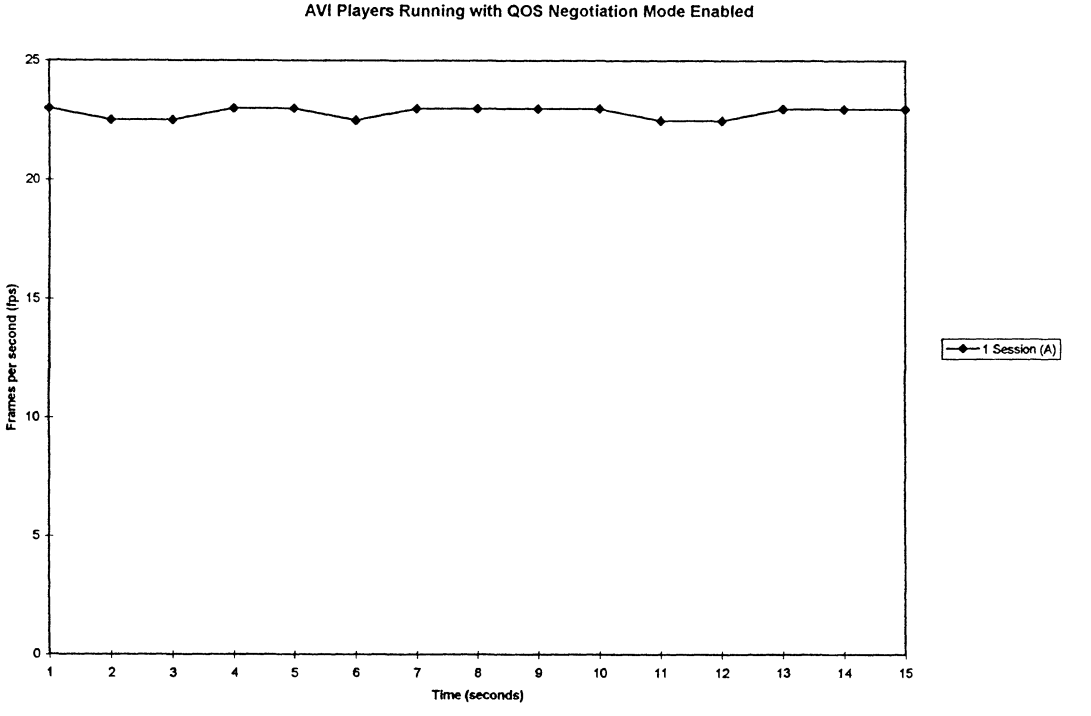

Figure 5.14: A single session of the AVI player running with the QOS negotiation mode enabled.

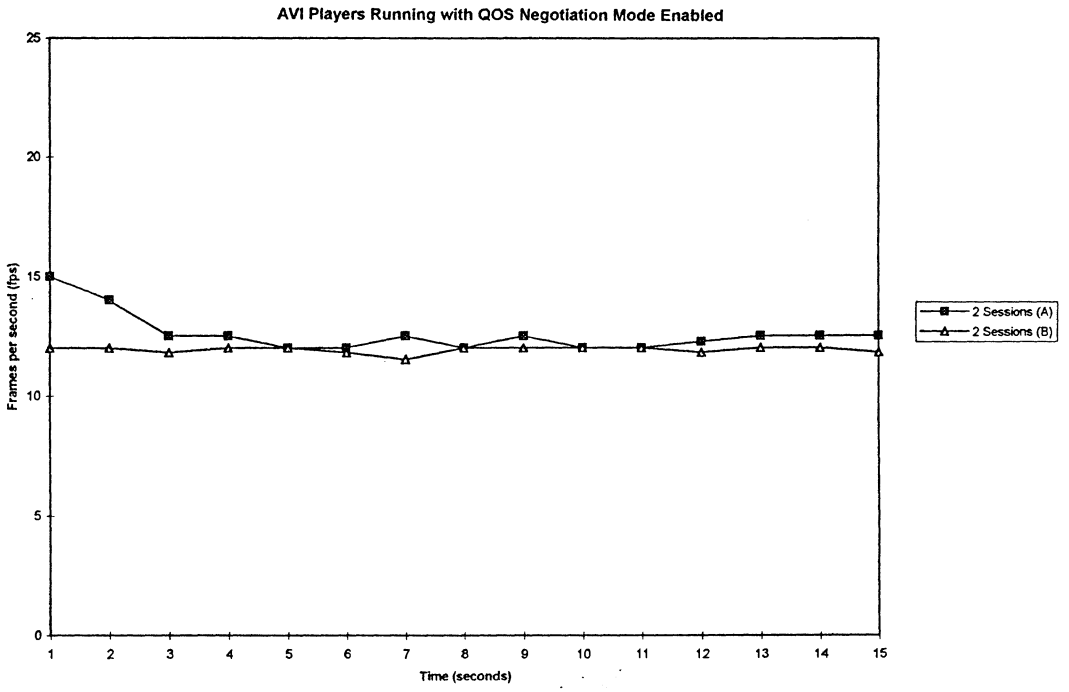

Figure 5.15: Two sessions of the AVI players running with the QOS negotiation mode enabled.

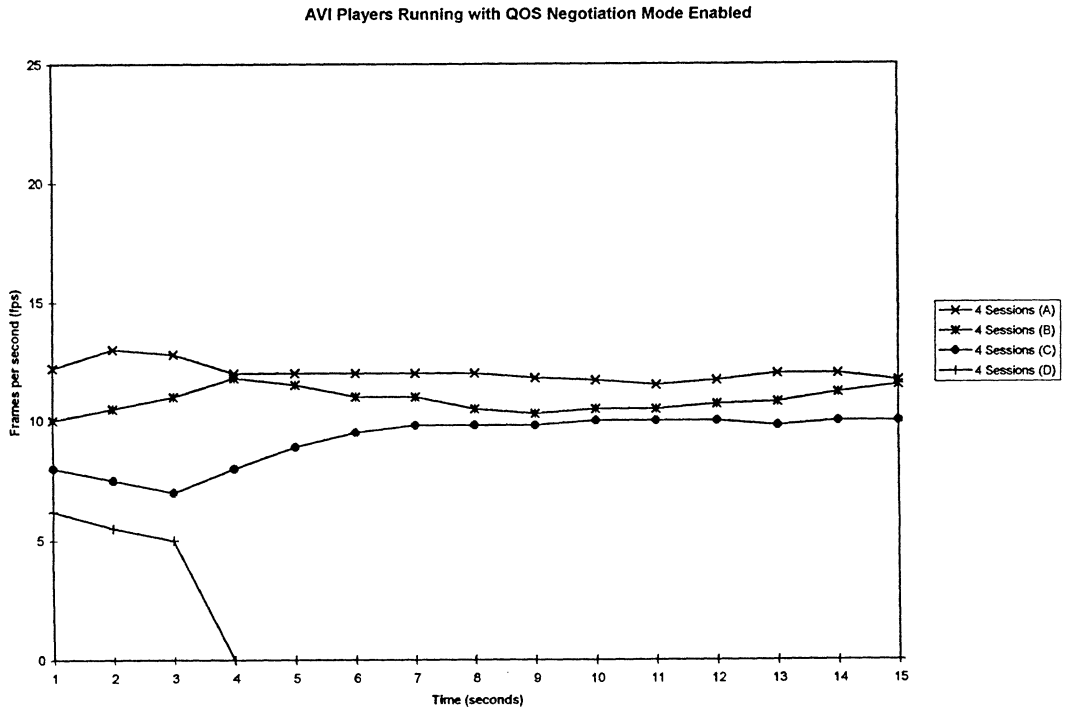

Figure 5.16: Four sessions of the AVI players running with the QOS negotiation mode enabled.

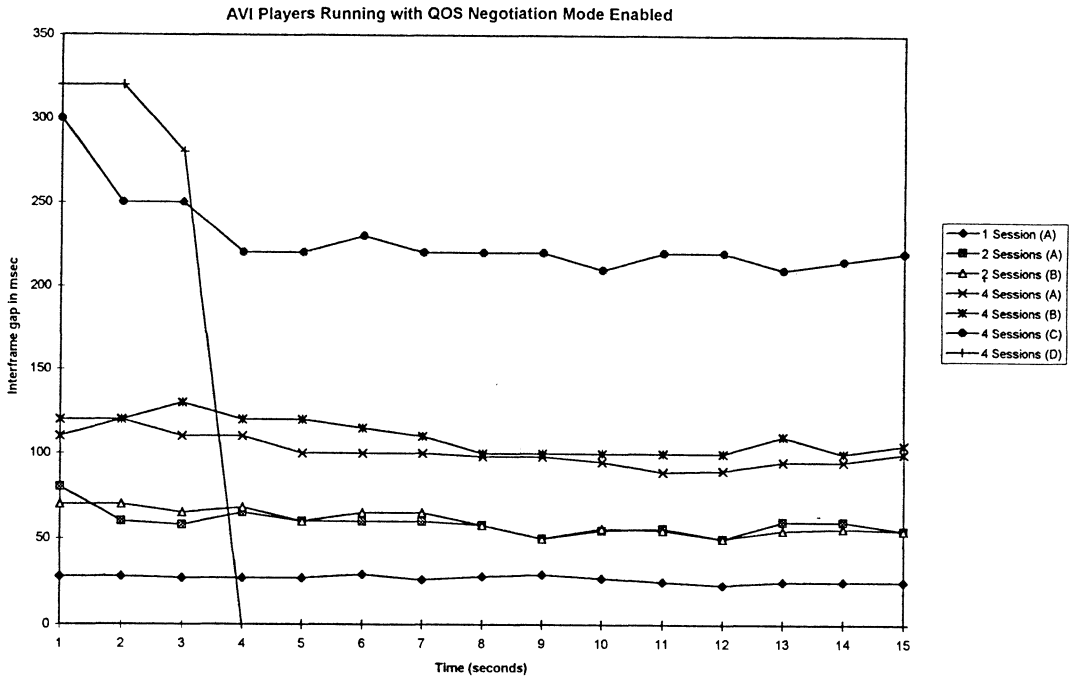

Figure 5.17: Interframe gaps measured by running the AVI players with the QOS negotiation mode enabled. Separate plots of the results are available for: One session (Figure 5.18), two sessions (Figure 5.19), and four sessions (Figure 5.20).

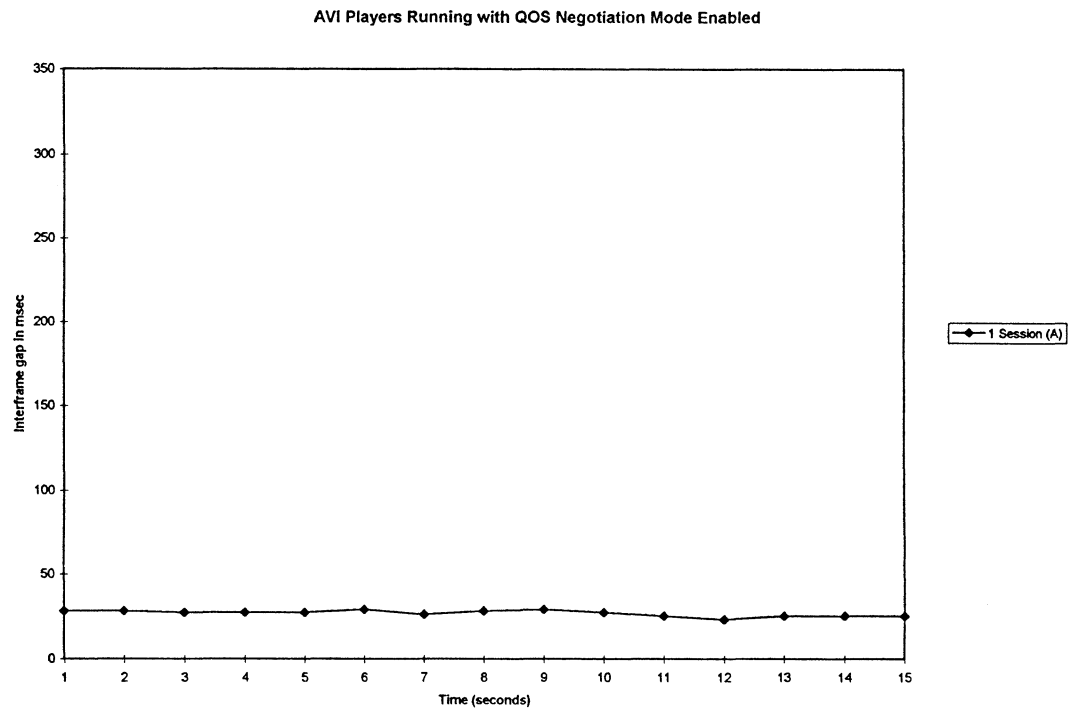

Figure 5.18: Interframe gaps measured by running a single session of the AVI player with the QOS negotiation mode enabled.

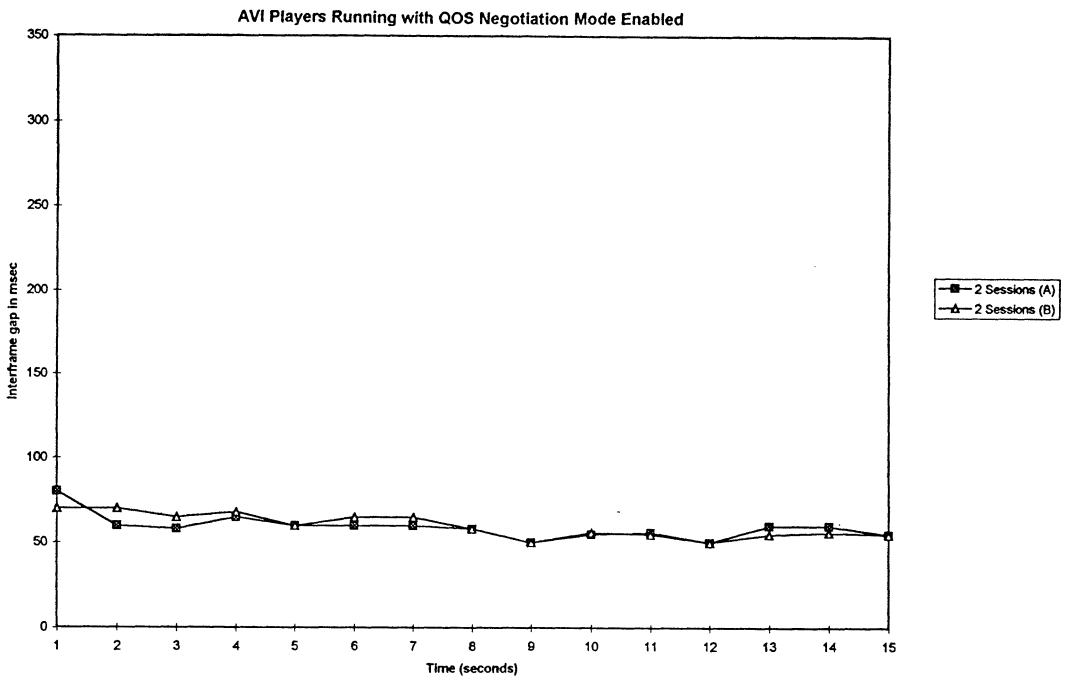

Figure 5.19: Interframe gaps measured by running two sessions of the AVI players with the QOS negotiation mode enabled.

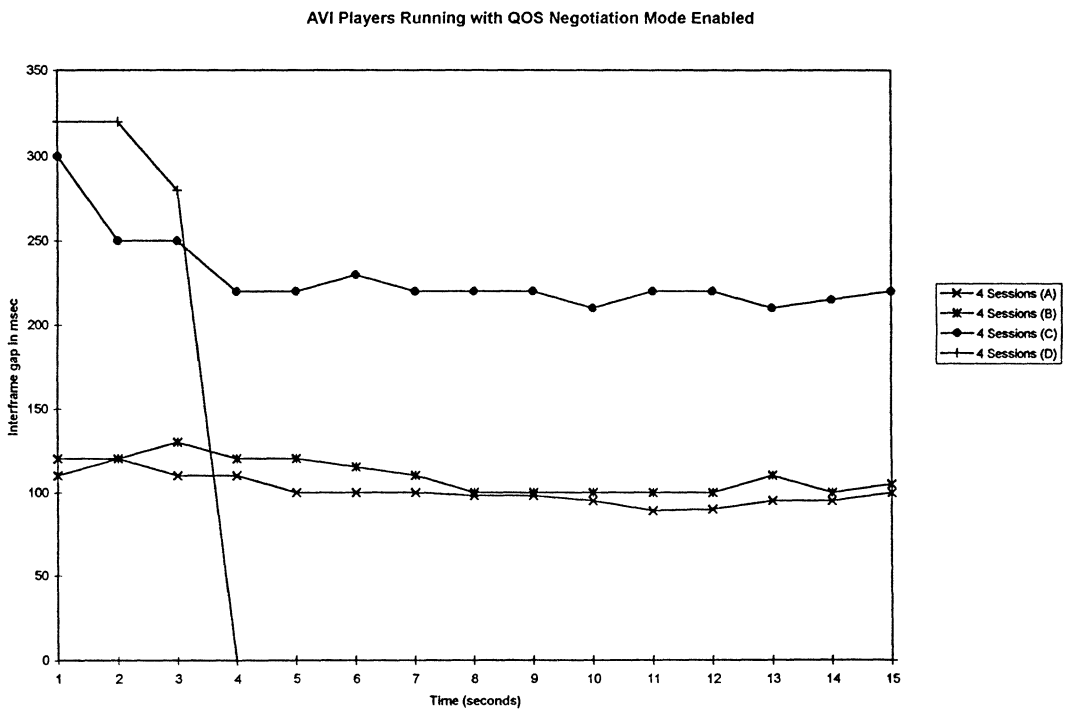

Figure 5.20: Interframe gaps measured by running four sessions of the AVI players with the QOS negotiation mode enabled.

To investigate the effects of latency caused by hard drive access, we run the above tests with one condition altered. We make all the sessions sharing the same AVI file. The results are shown in Figure 5.21,with QOS negotiation disabled, and in Figure 5.25, with QOS negotiation enabled. In both situations, the performance is better than the case with accessing four separate AVI movies. This can be explained by the disk caching facilities provided by Windows NT. Since a portion of the AVI file is available in cache memory, when another session tries to play the AVI file, it may find a hit in the cache and read it from the cache memory instead of from the much slower hard drive. This is consistent with the findings in the references [47,48].

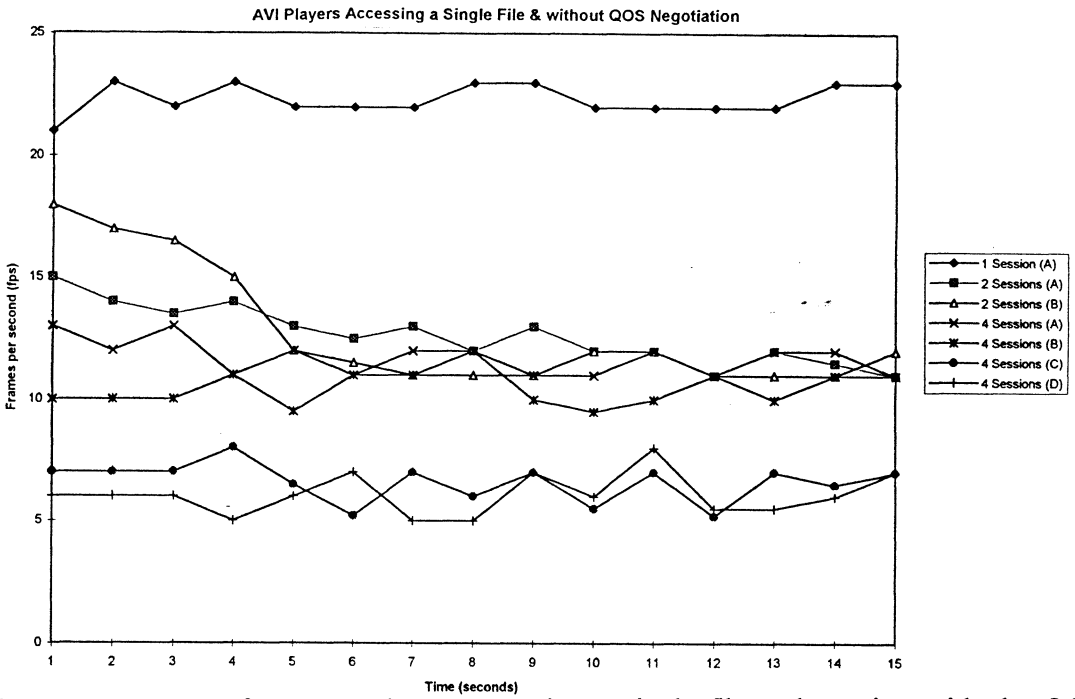

Figure 5.21: Up to four AVI players accessing a single file and running with the QOS negotiation mode disabled. Separate plots of the results are available for: One session (Figure 5.22), two sessions (Figure 5.23), and four sessions (Figure 5.24).

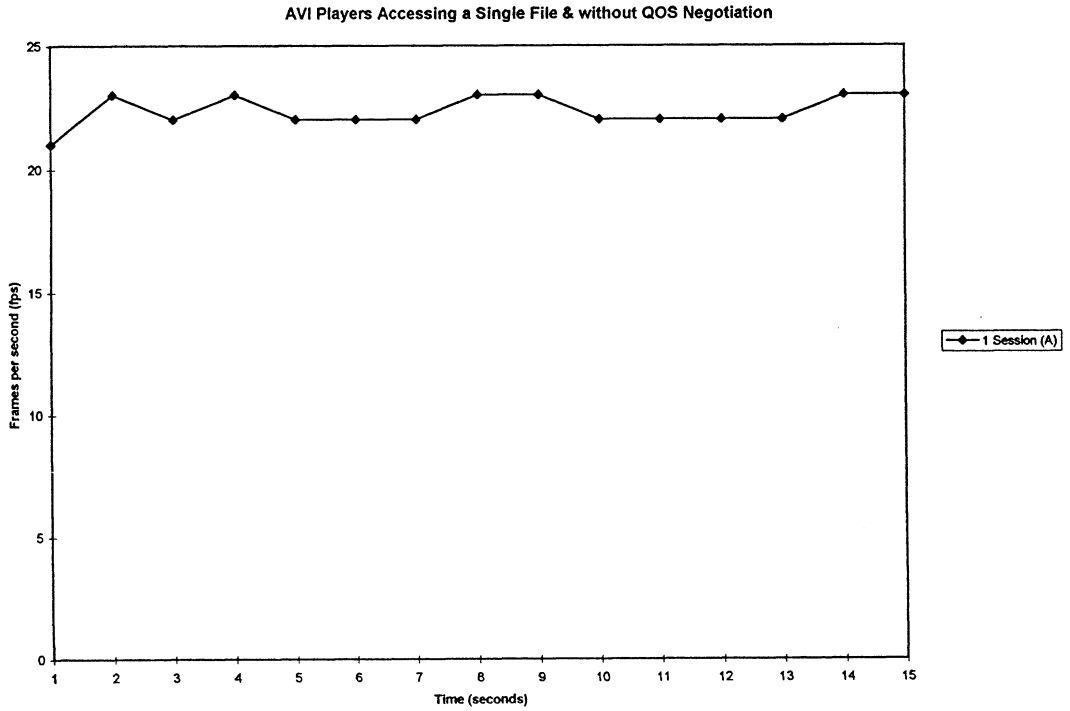

Figure 5.22: A single session of the AVI player running with the QOS negotiation mode disabled.

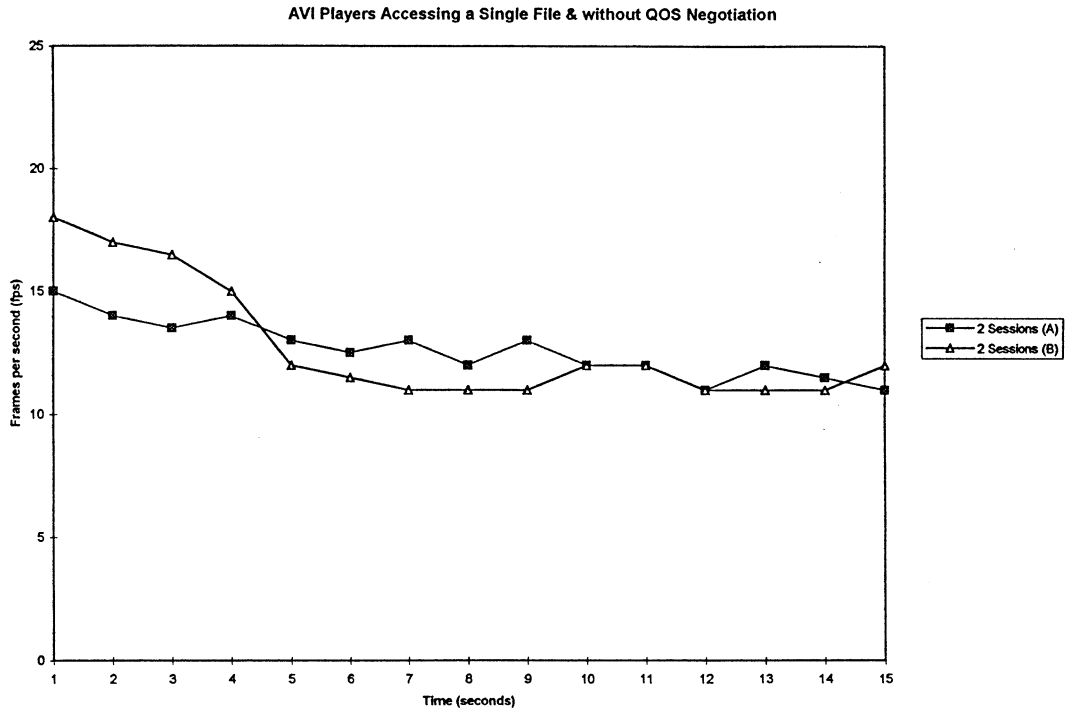

Figure 5.23: Two sessions of the AVI players accessing a single file and running with the QOS negotiation mode disabled.

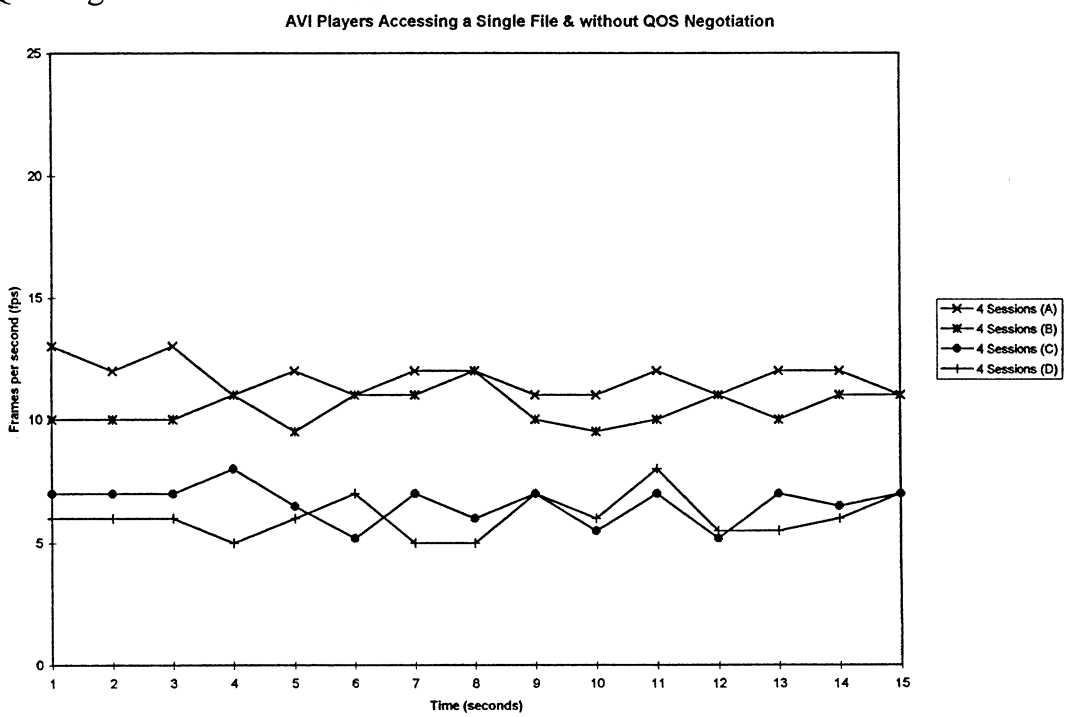

Figure 5.24: Four sessions of the AVI players accessing a single file and running with the QOS negotiation mode disabled.

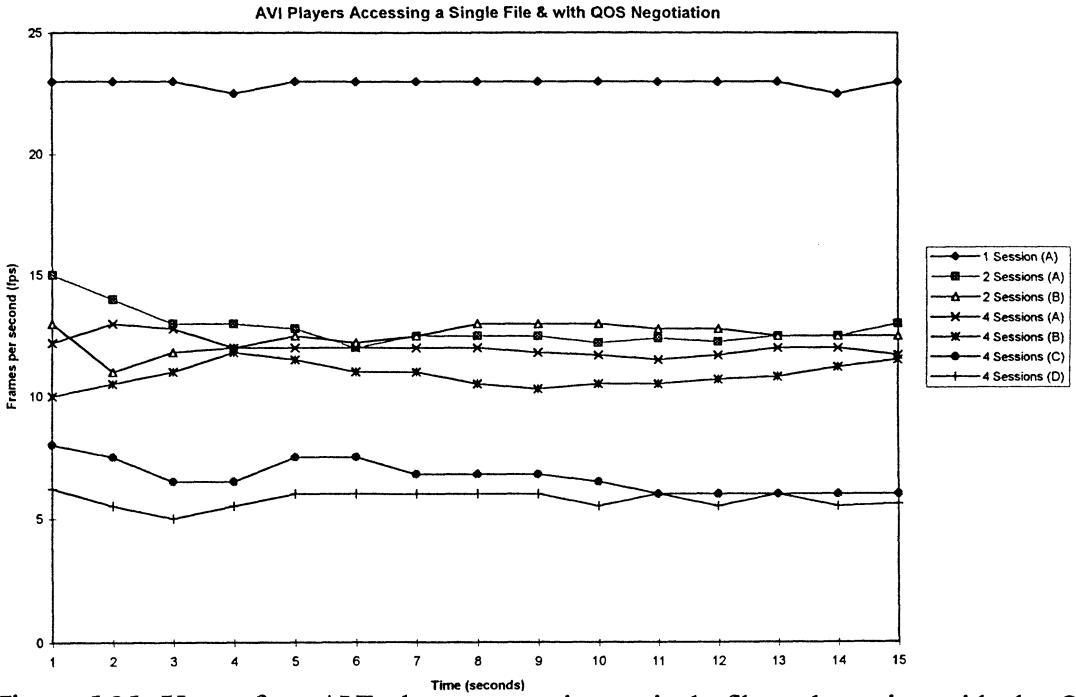

Figure 5.25: Up to four AVI players accessing a single file and running with the QOS negotiation mode enabled. Separate plots of the results are available for: One session (Figure 5.26), two sessions (Figure 5.27), and four sessions (Figure 5.28).

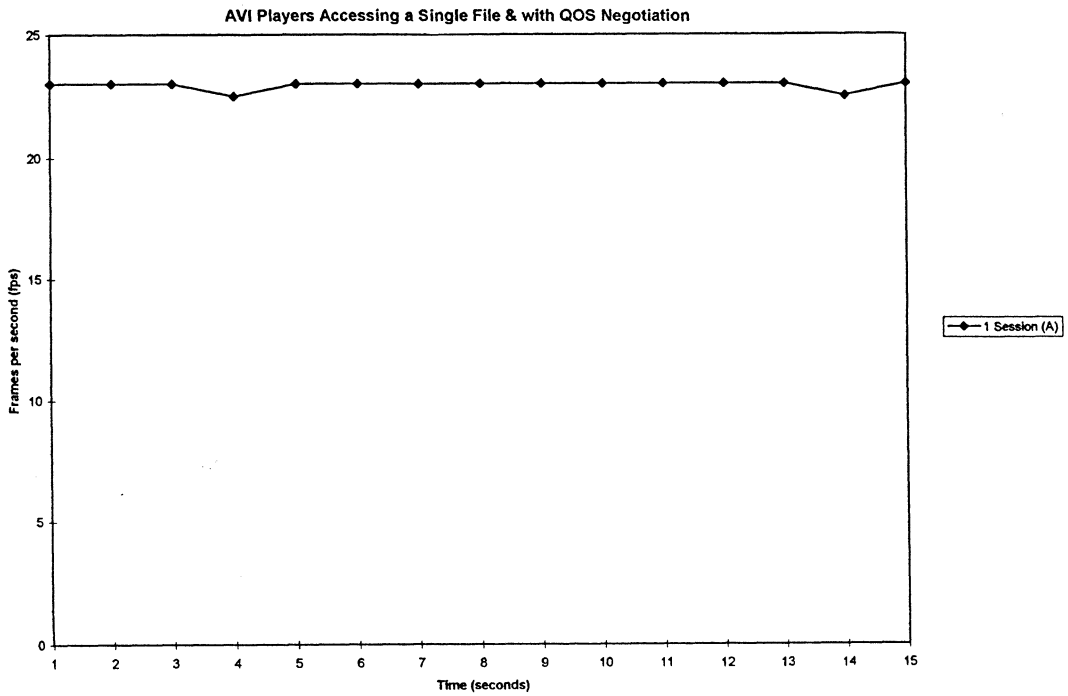

Figure 5.26: A single session of the AVI player running with the QOS negotiation mode enabled.

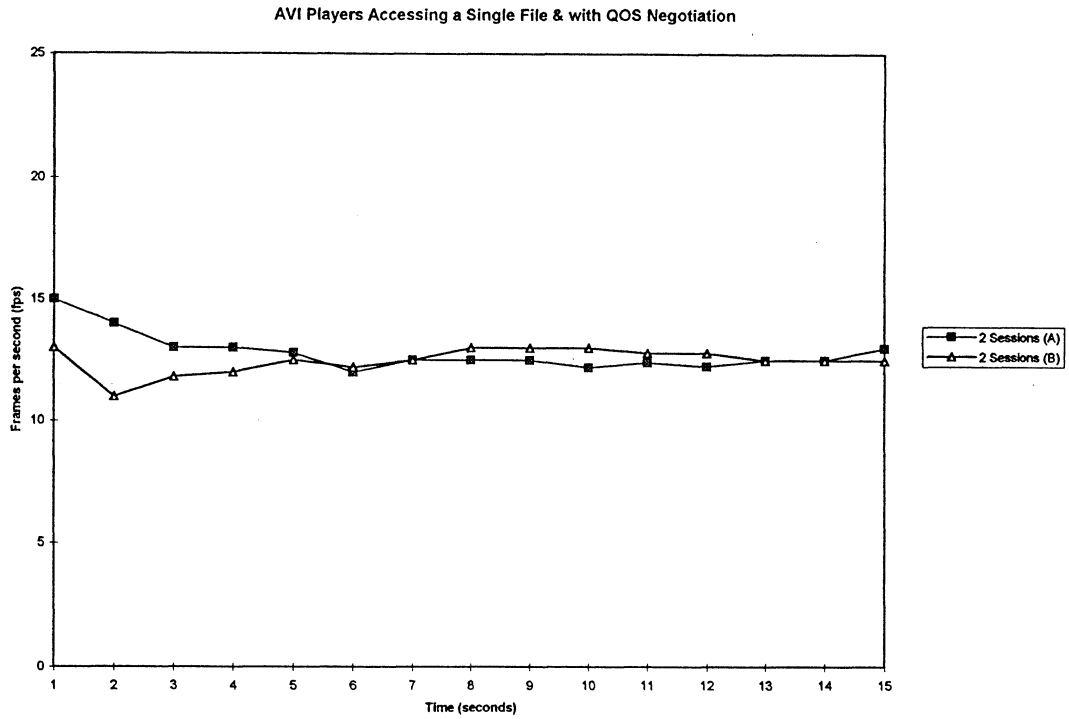

Figure 5.27: Two sessions of the AVI players accessing a single file and running with the QOS negotiation mode enabled.

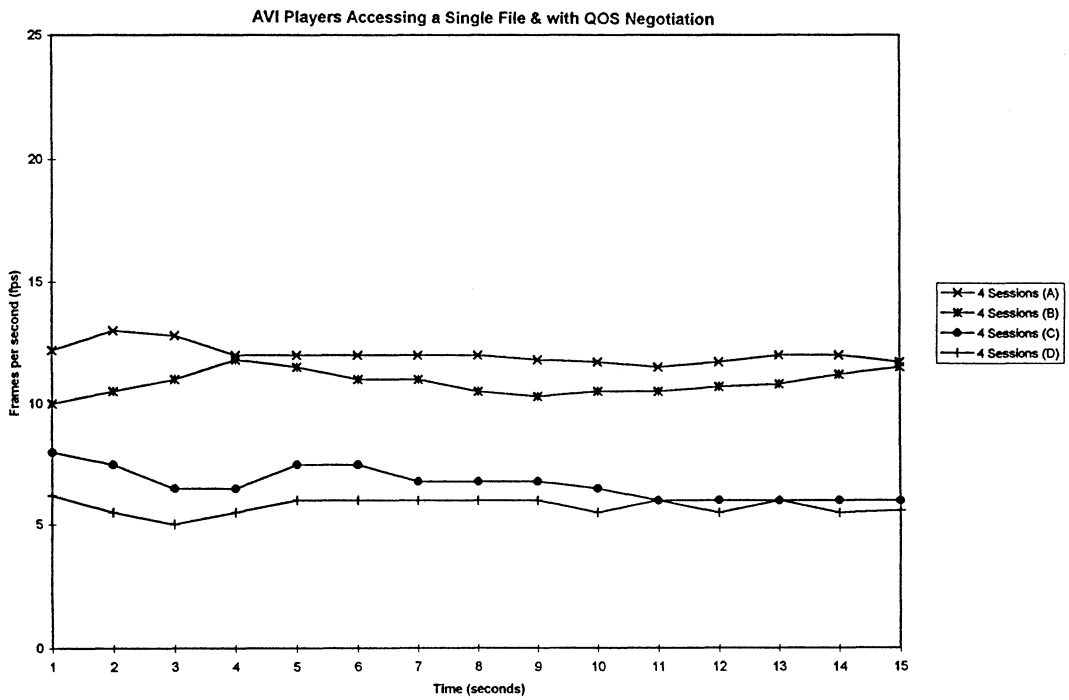

Figure 5.28: Four sessions of the AVI players accessing a single file and running with the QOS negotiation mode enabled.

Figures 5.29 and 5.30 show the results of a test where QOS negotiation mode is enabled only for the sessions running on machine *C* and machine *D*. The two more powerful machines (*A* and *B*) are running the sessions without any QOS management. The results indicate that while the machines *C* and *D* have less resources (slower processors and less RAM), the amount of interframe gap is less when QOS negotiation mode is enabled. This test demonstrates the importance of QOS management in multimedia applications.

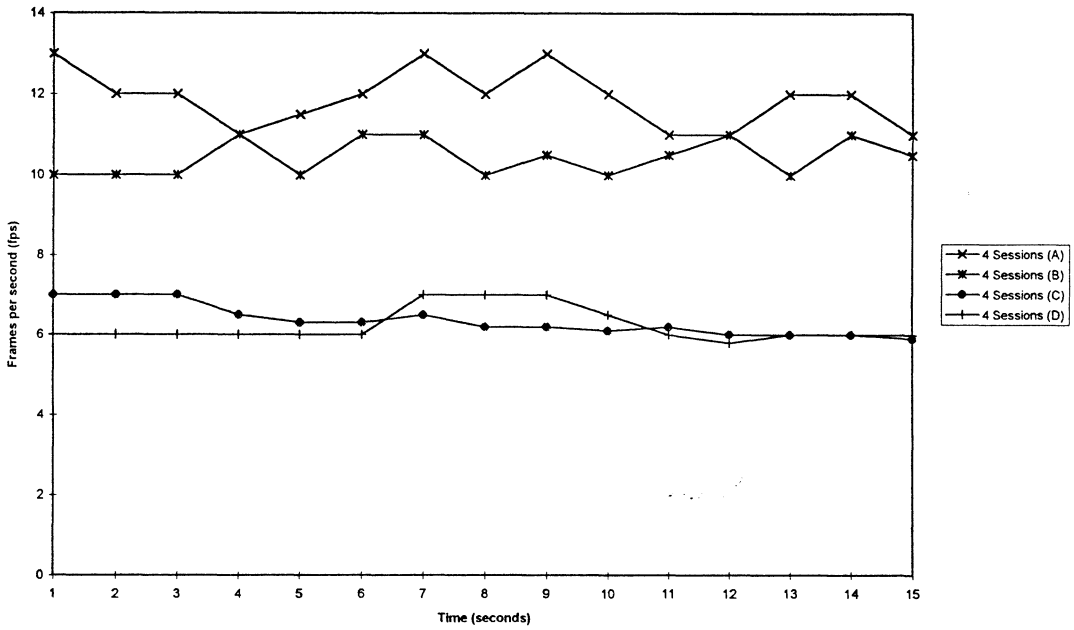

Figure 5.29: Frame rate versus time: *Machines A* and *B* are running without QOS management while *Machines C* and *D* running in QOS mode.

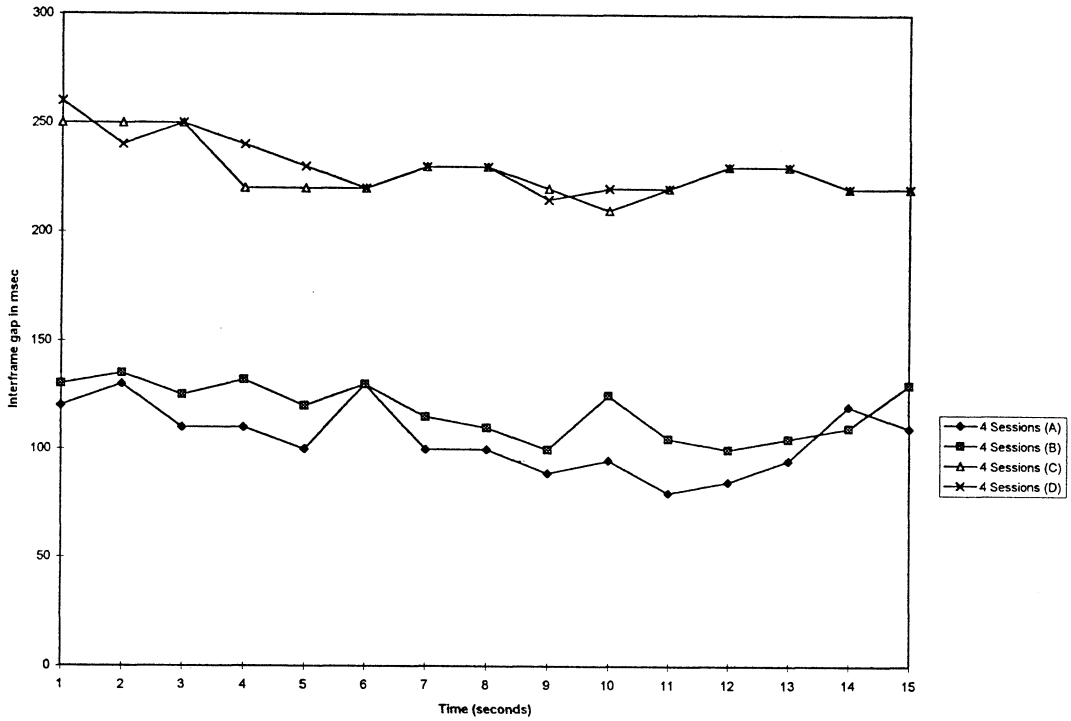

Figure 5.30: Interframe gap versus time: *Machines A* and *B* are running without QOS management while *Machines C* and *D* running in QOS mode.

The last experiment is designed to evaluate the effectiveness of dynamic QOS control. For this experiment we have added two additional machines (*E* and *F*) that are equipped the same way as the less powerful machines (*C* and *D*). The same 160 pixels by 120 pixels by 256 colors AVI file is used here since we want to keep the temporal resolution constant for each session. However, we allow each session to request its own frame rate before the sessions begin. Machine *A* requests 22 fps as the preferred rate, 10 fps as the acceptable rate, and 8 fps as the

unacceptable rate. Machine ***B*** requests 15 fps as the preferred rate, 13 fps as the acceptable rate, and 9 fps as the unacceptable rate. Machine ***C*** requests 12 fps as the preferred rate, 9 fps as the acceptable rate, and 7 fps as the unacceptable rate. Machine ***D*** requests 15 fps as the preferred rate, 9 fps as the acceptable rate, and 5 fps as the unacceptable rate. Machine ***E*** requests 20 fps as the preferred rate, 10 fps as the acceptable rate, and 8 fps as the unacceptable rate. Sessions ***A***, ***B***, ***C***, ***D***, and ***E*** are set to start at the same time at time T_0. Five seconds after that, machine ***F*** begins a new session requesting 12 fps as the preferred rate, 8 fps as the acceptable rate, and 5 fps as the unacceptable rate. All sessions are executed with the QOS negotiation mode enabled.

The results are shown in Figure 5.31. Machine ***A*** is able to play the digital video file at its preferred rate of 22 fps for the first five seconds of the session, until machine ***F*** starts the AVI player. When machine ***F*** requests for a new session, the QOS negotiation agent tries to find an acceptable QOS level that both the network and the application can agree on. In order to accommodate machine ***F***'s request at 12 fps, the QOS orchestration service decides to degrade the session running by machine ***A*** from its preferred frame rate at 22 fps down to its acceptable frame rate at 10 fps. The change of the temporal resolution of session ***A*** is done by resetting the execution rate of that thread. All other sessions maintained their throughput in a stable manner throughout the test duration.

Although the machines *C, D,* and *E* do not get to play their sessions at their preferred rates they specified, they all performed within the acceptable tolerances. Jitters are more severe with the less powerful machines but still within the acceptable range as seen in Figure 5.32.

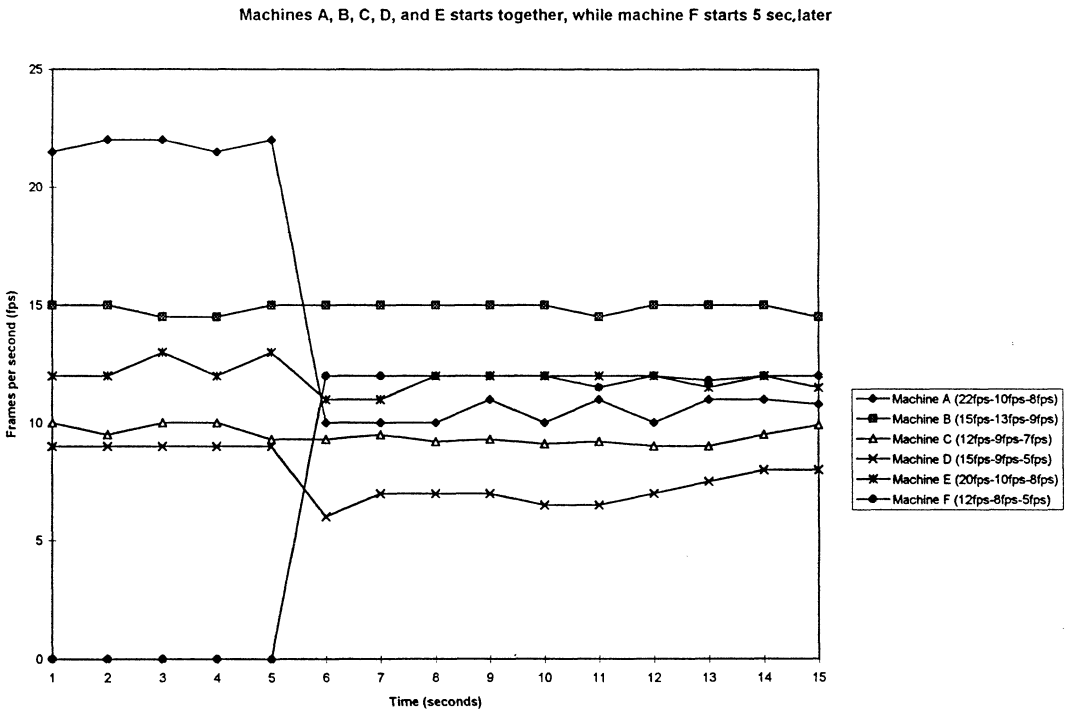

Figure 5.31: Frame rate versus time: Dynamic QOS control with QOS negotiation.

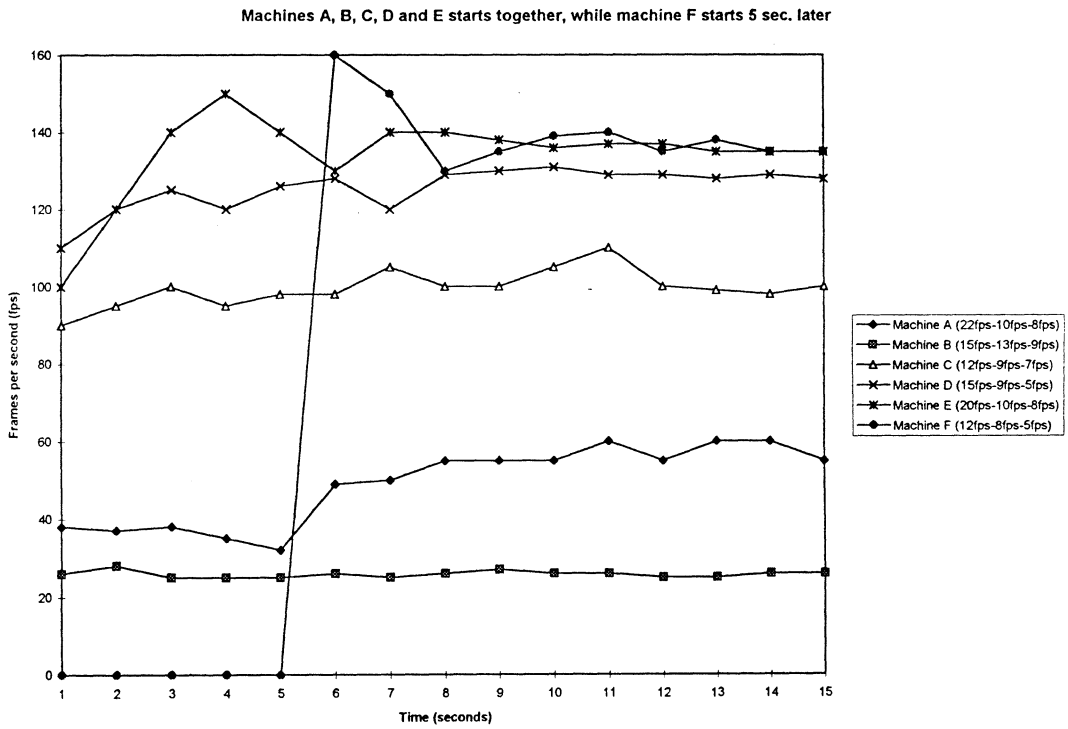

Figure 5.32: Iterframe gap versus time: Dynamic QOS control with QOS negotiation.

Chapter 6

Conclusions and Future Research

6.1 Concluding Remarks

This research has addressed the impact of application QOS requirements on synchronization problems that a multimedia network operating system must resolve. We have studied the temporal properties of continuous media types and identified operating system supports for distributed multimedia computing. This includes resource management support, architectural support, and programming support.

We have outlined an integrated QOS management model which maps application QOS parameters through all the layers of the entire system, from the application layer down to the network layer. This mapping process is done automatically throughout the application and the transport subsystems, thus protecting application programmers form the communication chores. However, the use of QOS parameterization of connections does not imply that a single fully generic transport protocol can cater for all types of multimedia traffic equally well. Instead, different types of media traffic require specialized protocols and control data. The layered architecture of the QOS management framework is

designed to support future expansions by adding additional protocols to the appropriate layer.

We have also discussed the use of the QOS negotiation agents to dynamically control QOS variations during a multimedia session. Experimental results have demonstrated the effectiveness of this negotiation scheme which adjusts QOS levels according to the system resources and the application requirements.

Multithreaded programming experiments have indicated that the operating system (OS) is a vital component in building an effective multimedia applications platform. Continuous media communication would be impractical without multithreading and preemptive scheduling facilities in an OS.

We envision that with true integrated QOS support from the next generation multimedia network operating systems, less powerful (low-cost) computers will perform well in delivering multimedia services since system resources will be better managed and utilized.

6.2 Future Research Directions

The ability to build better distributed multimedia applications depends on much more than the operating system itself. A number of issues remains to be investigated:

1. In this research, we have limited our discussion to end-to-end communication in a local area networking environment. However, the proposed layered architecture should be general enough to be adopted in wide are networks (WAN) or other gigabit network technologies such as Asynchronous Transfer Mode (ATM) networks [49].

2. The number of QOS dimensions in the architecture can be expanded to include support for security transactions and cost functions based on usage. This kind of support is important for doing business over the WAN or the Internet.

3. The integrated QOS management approach suggests that Ethernet is not effective in delivering multimedia services since it does not support QOS management schemes that are based on bandwidth reservation. In contrast, ATM networks have QOS characterization built into its architecture [50, 51]. It may be beneficial to incorporate other network

technologies such as ATM, 100VG-AnyLAN [52], and iso-Ethernet [53] into our proposed model.

4. The area of network support for rate-based flow control remains an important issue which we have not addressed. The effectiveness of TCP/IP in multimedia communication should also be further studied [54].

5. Currently, there are no multicasting capabilities built into the QOS negotiation agent or the end-point transmission protocols. It is possible to employ IP multicast in a future revision of the design.

6. A potential weakness in our QOS negotiation model is that too much time may be spent on negotiating between the layers and the remote supplier for an acceptable QOS level. It is also a concern that QOS levels may be modified too often when multimedia applications are competing for resources from a heavily loaded network.

7. Further research is needed in developing real-time support for existing operating systems. The microkernel architecture [55, 56, 57] in research operating systems such as the Real-Time Mach [58] is especially attractive in supporting real-time multimedia applications.

References

1. J. A. Adam, "Special Report/Multimedia: Applications, Implications," *IEEE Spectrum*, Vol. 30, No. 3, 1993, pp. 22-31.

2. J. Bach, S. Paul, and R. Jain, "A Visual Information Management System for Interactive Retrieval of Faces," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 4, 1993, pp. 619-628.

3. Y. H. Chang et al., "An Open-Systems Approach to Video on Demand," *IEEE Communications*, Vol. 32, No. 5, May 1994, pp. 68-80.

4. N. Dimitrova and F. Golshani, "Rx for Semantic Video Database Retrieval," *Proc. ACM Multimedia '94*, ACM Press, San Francisco, October 1994, pp. 219-226.

5. J. Sutherland and L. Litteral, "Residential Video Services," *IEEE Comm.*, Vol. 30, No. 7, July 1992, pp. 36-41.

6. T. Little and D. Venkatesh, "Prospects for Interactive Video-on-demand," *IEEE MultiMedia*, Vol. 1, No. 3, 1994, pp. 14-24.

7. W. Mackay and G. Davenport, "Virtual Video Editing in Interactive Multimedia Applications," *Comm. ACM*, Vol. 32, No. 7, 1989, pp. 802-810.

8. E. C. Chung and M. Celenk, "Novell Netware Multimedia Communication System Using Microsoft Windows," *Proc. 27th Southeastern Symposium on System Theory*, Starkville, Mississippi, March 1995, pp. 397-401.

9. B. Szuprowicz, *Multimedia Networking*, McGraw-Hill, 1995, p. 112.

10. B. Szuprowicz, *Multimedia Networking*, McGraw-Hill, 1995, p. 114.

11. M. Celenk and Y. Wang, "Distributed Computation in Local Area Networks of Workstations," *Parallel Algorithms and Applications*, Vol. 5, 1995, pp. 79-106.

12. J. Buford, *Multimedia Systems*, Addison-Wesley, 1994.

13. K. Nahrstedt and J. Smith, "The QOS Broker," *IEEE MultiMedia*, Spring 1995, pp. 53-67.

14. R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*, Prentice Hall, 1995, p. 571.

15. R. Steinmetz, " Synchronization Properties in Multimedia Systems," *IEEE Journal on Selected Areas in Comm.*, Vol. 8, No. 3, 1990, pp. 401-412.

16. T. Little, A. Ghafoor, C. Chen, C. Chang, and P. Berra, "Multimedia Synchronization," *IEEE Data Eng. Bulletin*, Vol. 14, No. 3, 1991, pp. 26-35.

17. T. Little and A. Ghafoor, "Network Considerations for Distributed Multimedia Object Composition and Communication," *IEEE Network*, Vol. 4, No. 6, 1990, pp. 32-49.

18. D. Bulterman and R. van Liere, "Multimedia Synchronization and Unix," *Proc. 2nd Int'l. Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991, pp. 108-119.

19. Y. Doğanata and A. Tantawi, "Making Cost-Effective Video Server," *IEEE MultiMedia*, Winter 1994, pp. 22-30.

20. H. Vin, P. Zellweger, D. Swinehart, and P. Rangan, "Multimedia Conferencing in the Etherphone Environment," *IEEE Computer*, October 1991.

21. D. Ferrari et al., "Network Support for Multimedia - A Discussion of the Tenet Approach," Technical Report TR-92-072, International Computer Science Institute, November 1992.

22. M. Hayter, *A Workstation Architecture to Support Multimedia*, Ph.D. dissertation, St. John's College, University of Cambridge, 1993.

23. J. F. Adam et al., "Media-Intensive Data Communications in a Desk-Area Network," *IEEE Communications*, Vol. 32, No. 8, August 1994, pp. 60-67.

24. M. Buddhikot, G. Parulkar, and J. Cox, "Design of a Large-Sale Multimedia Server," *Proc. INET 94/ JNEC5*, 1994, pp. 663.1-663.10.

25. M. Hodges, R. Sasnett, and M. Ackerman, "A Construction Set for Multimedia Applications," *IEEE Software*, pp. 37-41, January 1989.

26. W. Mackay and G. Davenport, "Virtual Video Editing in Interactive Multimedia Applications," *Communications of the ACM*, pp. 802-806, July 1989.

27. M. Hodges, R. Sasnett and J. Harward, "Musings on Multimedia," *UNIX Review*, pp. 83-85, February 1990.

28. M. Hodges, R. Sasnett, and E. Schlusselberg, "AthenaMuse Data Description Language," CECI Report, MIT, 1992.

29. "The AthenaMuse 2 Functional Specification," CECI (MIT) Report, May 1992.

30. "The AthenaMuse2 Architecture," CECI (MIT) Report, August 1992.

31. W. J. Clark, "Multipoint Multimedia Conferencing," *IEEE Communications*, pp. 44-48, May 1992.

32. T. Little and J. F. Gibbon, "Management of Time-Dependent Multimedia Data," *Proc. SPIE Symposium OE/FIBERS 1992, Enabling Technologies for Multi-Media, Multi-Service Networks*, September 1992.

33. B. Furht, "Multimedia Systems: An Overview," *IEEE Multimedia*, Vol. 1, No. 1, Spring 1994, pp. 47-59.

34. R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*, Prentice Hall, 1995, p. 615-635.

35. G. S. Blair, F. Garcia, D. Hutchison, and W. D. Shepherd, "Towards New Transport Services to Support Distributed Multimedia Apllications," *Multimedia '92: 4th IEEE COMSOC International Workshop*, Monterey, California, April 1-4, 1992.

36. H. Leopold, G. Blair, A. Campbell, G. Coulson, P. Dark, F. Garcia, D. Hutchison, N. Singer, and N. Williams, "Distributed Multimedia Communications System Requirements," OSI95/Deliverable ELIN-

1/P/V3, Alcatel ELIN Research, A-1210 Vienna, Ruthnergasse 1-7, Austria, April 1992.

37. A. Campbell, G. Coulson, and D. Hutchison, "A Suggested QOS Architecture for Multimedia Communications," ISO/IEC JTC1/SC21/WG1 N1201, International Standards Organisation, UK, December 1991.

38. D. L. Tennenhouse, "Layered Multiplexing Considered Harmful," *Protocols for High-Speed Networks*, Elsevier Science Publishers B. V., North Holland, 1990.

39. R. G. Herrtwich and L. Delgrossi, "Beyond ST-II: Fulfilling the Requirements of Multimedia Communication," *Proc. 3rd International Workshop on Network Operating System Support for Digital Audio and Video*, November 1992.

40. H. Tokuda, Y. Tobe, S. Chou, and J. Moura, "Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network," *Proc. ACM SIGCOMM '92*, August 1992.

41. C. A. Nicolaou, "A Distributed Architecture for Multimedia Communication Systems," Technical Report 220, Computer Laboratory, University of Cambridge, May 1991.

42. R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*, Prentice Hall, 1995, p. 439.

43. R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*, Prentice Hall, 1995, p. 430.

44. A. Dumas, *Programming WinSock*, Sams Publishing, 1995.

45. T. Q. Pham and P. K. Garg, *Multithreaded Programming with Windows NT*, Prentice Hall, 1996, pp. 17-22.

46. A. M. Van Tilborg and G. M. Koob, eds., *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kulwer Academic Publisher, Norwell, Mass., 1991.

47. H. J. Chen and T. Little, "Physical Storage Organizations for Time-Dependent Multimedia Data," *Proc. Foundations of Data Organization and Algorithms Conference*, October 1993 .

48. S. Ghandeharizadeh and C. Shahabi, "On Multimedia Repositories, Personal Computers, and Hierarchical Storage Systems," *Proc. ACM Multimedia 94*, San Francisco, pp. 407-416, October 1994.

49. D. Ferrari, "Distributed Delay Jitter Control in Packet-Switching Internetworks," Technical Report, International Computer Science Institue, Berkely, CA.

50. L. A. Crutcher and A. G. Waters, "Connection Management for an ATM Network," *IEEE Network*, Vol. 6, No. 6, pp. 42-55, Nov. 1992.

51. J. Jung and D. Seret, "Translation of QOS Parameters into ATM Performance Parameters in B-ISDN," *Proc. Infocom 93*, Vol. II, IEEE, New York, pp. 748-755, 1993.

52. D. Newman and B. Levy, "100Base-T vs. 100VG: The Real Fast Ethernet," *Data Communications*, Vol. 25, No. 3, pp. 66-80, March 1996.

53. D. Greenfield, "Iso-Ethernet: A Reprieve for Ethernet?" *Data Communications*, Vol. 25, No. 3, pp. 115-120, March 1996.

54. C. Papadopoulos and G. M. Parulkar, "Experimental Evaluation of SUNOS IPC and TC/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, Vol. 1, No. 2, pp. 199-216, April 1993.

55. M. J. Accetta, et al., "Mach: A New Foundation for UNIX develpoemnt," *Proc. USENIX Conference*, July 1986.

56. D. L. Black, et al., "Microkernel Operating System Architecture and Mach," *Proc. Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

57. D. Golub, et al., "UNIX as an Application Program," *Proc. Summer USENIX Coference*, June 1990.

58. H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictible Real-Time System," *Proceedings of USENIX Mach Workshop*, October 1990.

Appendix A

Audio/Video Interleaved Digital Video Format

The experiments conducted in this research make use of Video for Windows (VfW) which is a digital video framework developed by Microsoft. Video for Windows provides a programming interface for video playback and recording and specifies the Audio/Video Interleaved (AVI) file format for storing digital video and audio data. The AVI format is a RIFF (Resource Interchange File Format) file specification, jointly developed by IBM and Microsoft, to be used with applications that capture, edit, and playback audio/video sequences. In general, AVI files may contain multiple streams of different types of data. Most AVI sequences store both audio and video streams. There is also a variation for the AVI sequence that contains only video data and does not require an audio stream. Specialized AVI sequences may include a control track or a Musical Instrument Digital Interface (MIDI) track as an additional data stream. The control track can be used to control external devices such as a media control interface (MCI) videodisc player. The Appendix A discusses the following topics as related to the AVI file format:

1. Required chunks of an AVI file,
2. optional chunks of an AVI file, and
3. developing routines to write AVI files.

Detailed information on AVI and RIFF file formats are available from the Microsoft Developers' Network.

RIFF files are built from *chunks*, each of which consists of a four-character *chunk type*, followed by an integer value indicating the amount of data in the chunk, and then the actual data. Since chunks can contain other chunks, thus RIFF files have a hierarchical structure. The root chunk has 'RIFF' as its chunk type and the first four bytes of the data field are reserved for a *form type*. Similar to chunk type, form type is a four-character identifier to specify the structure of the embedded data. The RIFF file specification is extensible since new media types can be accommodated by introducing new chunk types or form types. Figure A.1 shows a list of chunk and form types supported by RIFF.

Chunk type or form type	Data type contained in the chunk
AVI	An audio/video interleaved sequence
INFO	Information about the file including creation date, copyright holder, and comments
LIST	A list of subchunks
PAL	A color palette
RDIB	A device-independent bitmap (DIB) image
RMID	A Musical Instrument Digital Interface (MIDI) sequence
RTF	Rich Text Format (RTF) including text and graphics
WAVE	Waveform audio samples

Figure A.1: Different RIFF chunk types and form types.

AVI files use the AVI RIFF form which is identified by the chunk type code 'AVI '. All AVI files include two mandatory LIST chunks, which define the format of the streams and stream data. An index chunk may also be included in AVI files, which specifies the location of data chunks within the file. The index chunk is useful for editing and keeping track of the frames in the file. An AVI file with these components has the following form:

```
1:   RIFF ('AVI '
2:     LIST ('hdrl'
3:           . . .
4:                  )
5:      LIST ('movi'
6:           . . .
7:                  )
8:      ['idx1'<AVI Index>]
9:                  )
```

The LIST chunks and the index chunk are subchunks of the RIFF 'AVI ' chunk. The 'AVI ' chunk type identifies the file as an AVI RIFF file. The LIST "hdrl" chunk defines the format of the data and it is the first required list chunk. The LIST "movi" chunk contains the digital video data for the AVI

sequence and is the second required list chunk. The "idx1" chunk is the optional index chunk that contains location information of the playback frames. These three components must be organized in the proper sequence within the AVI file.

The LIST "hdrl" and LIST "movi" chunks use subchunks for their data. The following example shows the AVI RIFF form, expanded with the chunks required to complete the LIST "hdrl" and LIST "movi" chunks:

```
1:   RIFF ('AVI '
2:           LIST ('hdrl'
3:                 'avih'(<Main AVI Header>)
4:                 LIST ('strl'
5:                      'strh'(<Stream header>)
6:                       'strf'(<Stream format>)
7:                       'strd'(additional header data)
8:                           . . .
9:                                )
10:                          . . .
11:            )
12:  LIST ('movi'
13:                {SubChunk | LIST('rec '
14:                                   SubChunk1
```

```
15:                                    SubChunk2
16:                                      . . .
17:                                           )
18:                                      . . .
19:                 }
20:                                      . . .
21:             )
22:           ['idx1'<AVIIndex>]
23:  )
```

In an AVI file, the main header is identified with the four-character identifier code "avih". The header contains general information about the file, including the number of streams within the file, and the height and width of the AVI sequence. The data structure of the main header is defined as follows:

```
1:   typedef struct {
2:        DWORD  dwMicroSecPerFrame;
3:        DWORD  dwMaxBytesPerSec;
4:        DWORD  dwReserved1;
5:        DWORD  dwFlags;
6:        DWORD  dwTotalFrames;
7:        DWORD  dwInitialFrames;
8:        DWORD  dwStreams;
```

```
9:        DWORD  dwSuggestedBufferSize;
10:       DWORD  dwWidth;
11:       DWORD  dwHeight;
12:       DWORD  dwScale;
13:       DWORD  dwRate;
14:       DWORD  dwStart;
15:       DWORD  dwLength;
16:   } MainAVIHeader;
```

The dwMicroSecPerFrame field specifies the period between video frames, which is used to specify the overall timing for the file.

The dwMaxBytesPerSec field specifies the approximate maximum data rate of the file. This value indicates the number of bytes per second that the system must handle to present an AVI sequence as specified by the other parameters contained in the main header and stream header chunks.

The dwFlags field contains any one of the following flags for the file:

AVIF_COPYRIGHTED

Indicates the AVI file contains copyrighted data. When this flag is set, applications should not let user duplicate file or the data in the file.

AVIF_HASINDEX

Indicates the AVI file has an "idx1" chunk.

AVIF_ISINTERLEAVED

Indicates the AVI file is interleaved. The computer system can stream interleaved data from a CD-ROM more efficiently than non-interleaved data.

AVIF_MUSTUSEINDEX

Indicates the index should be used to determine the order of presentation of the data. When this flag is set, it implies the physical ordering of the chunks in the file that does not correspond to the presentation order.

AVIF_WASCAPTUREFILE

Indicates the AVI file is a specially allocated file used for capturing real-time video. Typically, capture files have been defragmented by user so that video capture data can be efficiently streamed into the file. If this flag is set, an application should warn the user before writing over the file with this flag.

The dwTotalFrames field of the main header specifies the total number of frames of data in file.

The dwInitialFrames is used for audio/video interleaved files. When creating interleaved files, the number of frames in the file prior to the initial frame of the AVI sequence should be specified in this field.

The dwStreams field specifies the number of data streams in the file. For instance, a file with audio and video contains two data streams.

The dwSuggestedBufferSize field specifies the suggested buffer size for reading the file. In general, this buffer size should be large enough to contain the largest chunk in the file. If the field is set to the size of zero, or if the size is too small, the playback application must reallocate memory during playback which will reduce performance. For an interleaved file, the buffer size should be large enough to read an entire record and not just a chunk to avoid memory reallocation.

The dwWidth and dwHeight fields specify the width and height of the AVI file in pixels, respectively.

The dwScale and dwRate fields are used to specify the general time scale that the AVI file would use. In addition to the general time scale, each stream can have its own time scale. The time scale expressed in samples per second is determined by dividing dwRate by dwScale.

The dwStart and dwLength fields specify the starting time and the total length of the AVI file. The units are defined by the two fields dwRate and dwScale. The dwStart field is typically set to zero as the initial start time.

The main header is followed by one or more stream header ("strl") chunks. A "strl" chunk is required for each data stream and it contains information about each data stream in the file. The data structure of the stream header is defined as follows:

```
1:   typedef struct {
2:        FOURCC  fccType;
3:        FOURCC  fccHandler;
4:        DWORD   dwFlags;
5:        DWORD   dwReserved1;
6:        DWORD   dwInitialFrames;
7:        DWORD   dwScale;
```

```
8:        DWORD   dwRate;
9:        DWORD   dwStart;
10:       DWORD   dwLength;
11:       DWORD   dwSuggestedBufferSize;
12:       DWORD   dwQuality;
13:       DWORD   dwSampleSize;
14:   } AVIStreamHeader;
```

The stream header specifies the type of data the stream contains, such as video or audio, by using a four-character identifier code. The `fccType` field is set to "vids" if the stream it specifies contains video data. In the case of audio data, it is set to "auds".

The `fccHandler` field contains a four-character code describing the installable codec (compressor /decompressor) to be used with the data.

The `dwFlags` field contains any flags for the data stream:

`AVISF_DISABLED`

This flag indicates that the stream data should be rendered only when explicitly enabled by the user.

AVISF_VIDEO_PALCHANGES

This flag indicates that information for palette changes are included in the AVI file.

The dwInitialFrames data field is used for audio/video interleaved files. When creating interleaved files, this field is utilized to specify the number of frames in the file prior to the initial frame of the AVI sequence.

The remaining fields describe the playback characteristics of the media stream. These factors include the playback rate (dwScale and dwRate), the starting time of the sequence (dwStart), the length of the sequence (dwLength), the size of the playback buffer (dwSuggestedBuffer), an indicator of the data quality (dwQuality), and the sample size (dwSampleSize).

A stream format ("strf") chunk must follow a stream header ("strh") chunk. The format of the data in the stream is described by the stream format chunk. For video streams, the information in this chunk is a BITMAPINFO structure including palette information (e.g., 8-bit color). For audio streams, the information in this chunk is a WAVEFORMATEX or PCMWAVEFORMAT data structure. The WAVEFORMATEX structure is an extended version of the

WAVEFORMAT structure. The "strl" chunk might also contain a stream data ("strd") chunk. Whenever a stream data chunk is used, it always follows the stream format chunk. The format and content of this chunk are defined by installable codec drivers. Typically, codec drivers use this information for configuration. Multimedia applications that read and write RIFF files do not need to decode this information. The applications simply transfer this data to and from a codec driver as a memory block and the actual compression and decompression are performed by the codec driver itself.

An AVI player associates the stream headers in the LIST "hdrl" chunk with the stream data in the LIST "movi" chunk by using the order of the "strl" chunks. The first "strl" chunk applies to stream 0, the second applies to stream 1, and so forth. For instance, if the first "strl" chunk describes video data, then the video data is contained in stream 0. Naturally, if the second "strl" chunk describes the wave audio data, the wave audio data is contained in stream 1.

Following the header information is a LIST "movi" chunk that contains chunks of the actual data in the streams; i.e., the image frames and sounds themselves. The data chunks can reside directly in the LIST "movi" chunk or they may be grouped into "rec " chunks. The "rec " grouping implies that the

grouped chunks should be read from disk all at once. This is used only for files specifically interleaved to play from a CD-ROM.

Similar to all RIFF chunks, the data chunks contain a four-character code to identify the chunk type. The four-character code that identifies each chunk consists of the stream number and a two-character code that defines the type of information encapsulated in the chunk. In the case of a waveform (audio) chunk, it is identified by a two-character code "wb". However, if the waveform chunk is corresponding to the second LIST "hdrl" stream description, it would have a four-character code of "01wb".

It is unnecessary for the audio data chunks to contain any information about its format since all the format information is already included in the header. An audio data chunk using the ## in the format to represent the stream identifier has the following format:

```
1:    WAVE  Bytes    '##wb'
2:          BYTE     abBytes[];
```

Utilizing different codec drivers, video data can be compressed or uncompressed DIBs. An uncompressed DIB has `BI_RGB` specified for the

biCompression field in its associated BITMAPINFO structure. A compressed DIB has a value other than BI_RGB specified in the biCompression field. A data chunk for an uncompressed DIB contains RGB video data. These chunks are identified with a two-character code of "db" which stands for DIB bits. Data chunks for a compressed DIB are identified with a two-character code of "dc" which stands for DIB compressed. Neither one of the video data chunk contains any header information about the DIBs. An uncompressed DIB data chunk has the following form:

```
1:   DIB  Bits   '##db'
2:        BYTE   abBits[];
```

A compressed DIB data chunk has the following form:

```
1:   Compressed DIB   '##dc'
2:              BYTE  abBits[];
```

Video data chunks can also define new color palette entries used to update the palette during the playback of an AVI sequence. These chunks are identified with a two-character code of "pc" which stands for palette change. The color palette information has the following data structure:

```
1:   typedef struct {
2:       BYTE          bFirstEntry;
```

```
3:        BYTE          bNumEntries;
4:        WORD          wFlags;
5:        PALETTEENTRY  peNew;
6:    } AVIPALCHANGE;
```

The `bFirstEntry` field defines the first palette entry that needs to be changed and the `bNumEntries` field specifies the number of entries to be changed altogether. Where the `peNew` field contains the new color palette entries.

If a video stream includes any kind of color palette changes, the `AVITF_VIDEO_PALCHANGES` flag in the `dwFlags` field of the stream header should be set. This flag indicates that this video stream contains palette changes and warns the playback application that the application will have to handle the palette changes.

AVI files can also include an index chunk after the LIST "movi" chunk to provide location specific information to the AVI playback application. Essentially, the index chunk contains a list of the data chunks and their location in the file which provides efficient random access to the data within the file. If an index chunk is not included, the playback application would

have to search through the entire AVI file in order to locate a particular video frame or an audio sequence. An index chunk is especially effective for large AVI files.

The four-character identifier code for index chunks is "idx1". The data structure of an index entry has the following form:

```
1:   typedef struct {
2:       DWORD  ckid;
3:       DWORD  dwFlags;
4:       DWORD  dwChunkOffset;
5:       DWORD  dwChunkLength;
6:   } AVIINDEXENTRY;
```

The `ckid`, `dwFlags`, `dwChunkOffset`, and `dwChunkLength` entries are repeated in the AVI file for each indexed data chunk. If the file is interleaved, the index will also contain these entries for each "rec" chunk. The "rec" entries should have the `AVIIF_LIST` flag set and the list type in the `ckid` field.

The `ckid` field uses four-character codes for identifying the type of data chunk used. The `dwFlags` field specifies any one of the appropriate

flags for the data. The AVIIF_KEYFRAME flag indicates key frames in the video sequence. Key frames do not need previous video information to be decompressed. The AVIIF_NOTIME flag indicates a chunk that does not affect the timing of a video stream. For instance, changing palette entries indicated by a palette chunk should occur between displaying video frames. Thus, if an application needs to determine the length of a video sequence, it should not use chunks with the AVIIF_NOTIME flag. Under this condition, the application would ignore the palette chunk. The AVIIF_LIST flag indicates that the current chunk is a LIST chunk. The ckid field is then used to identify the type of LIST chunk.

The dwChunkOffset and dwChunkLength fields specify the position of and the length of the chunk. The dwChunkOffset field specifies the position of the chunk in the file relative to the 'movi' list.

The dwChunkLength field specifies the length of the chunk excluding the eight bytes required for the RIFF header.

The AVIF_HASINDEX in the dwFlags field of the AVI header is set when an index chunk is included in the RIFF file.

Appendix B

A WinSock C++ Class Library

Although the WinSock *C* library supplied by Microsoft provides the basic functions to create network applications, writing sophisticated networking protocols with the WinSock-specific *C* function calls is still a complex task. The WinSock *C++* class library introduced here is designed to provide object-oriented support for writing network applications efficiently. This *C++* class library is a class-wrapper of the original WinSock *C* library. It employs encapsulation, inheritance, and polymorphism to create an extensible framework for writing network applications. Many of the details required to utilize the WinSock functions are encapsulated in the class library. The result is a set of WinSock objects that make programming network functions more robust, concise, and maintainable. The remaining of Appendix B is a partial listing of the WinSock *C++* Class library.

```
////////////////////////////////////////////////////////////////////

//
// A C++ class library designed to manage WinSock network
// communication.
//
// This class library includes functions to:
// -  Initialize, get TCP/IP stack information, and clean up a
//    WINSOCK socket
// -  To create, send, receive, and destroy a stream socket
//
// The original WINSOCK library contains code copyrighted by
// Microsoft and Regents of the University of California (Berkeley).
// The class library is built on codes from Microsoft and Arthur
// Dumas, Programming WinSock, SAMS Publishing, 1994.
//
////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <memory.h>
#include "stdafx.h"
#include "clwinsok.h"

//////////////////////////////
// CL_WINSock  Class Library //
//////////////////////////////

////////////////////////////////////////////////////////////////////
// CL_WINSock constructor
//
// Constructs the CL_WINSock object. Initializes object member
// variables
//
////////////////////////////////////////////////////////////////////

CL_WINSock::CL_WINSock( WORD wVersionRequired )
{
  // initialize object member variables
  m_nLastError = NIL;
  m_wVersionRequired = wVersionRequired;
}


////////////////////////////////////////////////////////////////////
// CL_WINSock::Initialize()
//
// Initialize the WinSock sub-system.
////////////////////////////////////////////////////////////////////

int CL_WINSock::Initialize()
{
  int nStatus = CL_WINSOCK_NOERROR;

  m_nLastError = WSAStartup( m_wVersionRequired, &m_wsaData );

  if (m_nLastError != NoErr)
```

```
    nStatus = CL_WINSOCK_WINSOCK_ERROR;

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_WINSock::Close()
//
// Close down and clean up the WinSock sub-system.
////////////////////////////////////////////////////////////////////

int CL_WINSock::Close()
{
  int nStatus = CL_WINSOCK_NOERROR;

  if (WSACleanup() != NoErr)
  {
    m_nLastError = WSAGetLastError();
    nStatus = CL_WINSOCK_WINSOCK_ERROR;
  }

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_WINSock::GetInfo()
//
// Copy the WinSock TCP/IP stack information structure.
////////////////////////////////////////////////////////////////////

void CL_WINSock::GetInfo(LPWSADATA pwsaData)
{
  memcpy(pwsaData, &m_wsaData, sizeof(WSADATA));
}

////////////////////////////////
// CL_StreamSock  Class library //
////////////////////////////////

////////////////////////////////////////////////////////////////////
// CL_StreamSock constructor
//
// Constructs the CL_DatagramSock object.
// Initializes the object member variables.
////////////////////////////////////////////////////////////////////

CL_StreamSock::CL_StreamSock(CWnd *pParentWnd, UINT uMsg)
{
  // initialize the object member variables
  m_pParentWnd = pParentWnd;
  ASSERT(m_pParentWnd != NULL);
  m_uMsg = uMsg;
  ASSERT(m_uMsg != 0);
  InitVars();
```

```
}
////////////////////////////////////////////////////////////////////////
// CL_StreamSock destructor
////////////////////////////////////////////////////////////////////////

CL_StreamSock::~CL_StreamSock()
{
// Destroy the object when it's no longer in use.
}


////////////////////////////////////////////////////////////////////////
// CL_StreamSock::InitVars()
//
// Initialize the object member variables.
//
////////////////////////////////////////////////////////////////////////

void CL_StreamSock::InitVars(BOOL bInitLastError)
{
  if (bInitLastError)
    m_nLastError = NoErr;

  m_s = INVALID_SOCKET;
  memset(&m_sinLocal, 0, sizeof(SOCKADDR_IN));
  memset(&m_sinRemote, 0, sizeof(SOCKADDR_IN));
  m_bServer = FALSE;
}


////////////////////////////////////////////////////////////////////////
// CL_StreamSock::CreateSocket()
//
// To create a hidden window that can receive connection-oriented
// messages from WinSock. In addition, to create a socket and
// optionally bind it to a name if the socket is a server socket.
//
////////////////////////////////////////////////////////////////////////

int CL_StreamSock::CreateSocket(int nLocalPort)
{

  // if this version of the function is being called,
  // a valid port number must be specified
  if (nLocalPort <= 0)
    return CL_WINSOCK_PROGRAMMING_ERROR;


  // convert the port number into a string and
  // call the version of CreateSocket() which
  // accepts a string
  char pszLocalService[18];
  _itoa(nLocalPort, pszLocalService, 10);
  return CreateSocket(pszLocalService);
}
```

```
////////////////////////////////////////////////////////////////////
// CL_StreamSock::DestroySocket()
//
// To close the socket, clean up all queued data, and destroy the
// hidden window.
//
////////////////////////////////////////////////////////////////////

int CL_StreamSock::DestroySocket()
{
  int nStatus = CL_WINSOCK_NOERROR;

  // check if the socket is valid
  if (m_s == INVALID_SOCKET)
    nStatus = CL_WINSOCK_PROGRAMMING_ERROR;
  else
  {
    // remove any data in the write queue
    while (!m_listWrite.IsEmpty())
    {
      LPSTREAMDATA pStreamData =
               (LPSTREAMDATA)m_listWrite.RemoveHead();
      LPVOID pData = pStreamData->pData;
      delete pStreamData;

      m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_WRITE_ERR,
       (LPARAM)pData);
    }

    // remove any data in the read queue
    while (!m_listRead.IsEmpty())
    {
      LPSTREAMDATA pStreamData =
               (LPSTREAMDATA)m_listRead.RemoveHead();
      free(pStreamData->pData);
      delete pStreamData;
    }

    // close the socket and initialize variables
    closesocket(m_s);
    InitVars();

    // destroy the hidden window
    DestroyWindow();
  }

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_StreamSock::Connect()
//
// This version of the Connect() function takes a pointer to a
// string that represents the host name to send the data to and
```

```
// an integer that represents the port number to connect to.
//
////////////////////////////////////////////////////////////////////////

int CL_StreamSock::Connect(LPSTR pszRemoteName, int nRemotePort)
{
  // convert the port number into a string and then call the version
  // of Connect() which accepts a string service name or port number
  char pszRemoteService[18];
  _itoa(nRemotePort, pszRemoteService, 10);
  return Connect(pszRemoteName, pszRemoteService);
}

////////////////////////////////////////////////////////////////////////
// CL_StreamSock::Connect()
//
// This version of the Connect() function takes a pointer to a
// string that represents the host name to send the data to and
// an integer that represents the service name or port number to
// connect to.
//
////////////////////////////////////////////////////////////////////////

int CL_StreamSock::Connect(LPSTR pszRemoteName, LPSTR
pszRemoteService)
{
  LPHOSTENT   pHent;
  LPSERVENT   pSent;
  SOCKADDR_IN sinRemote;
  int         nStatus = CL_WINSOCK_NOERROR;

  while (TRUE)
  {
    // assign the address family
    sinRemote.sin_family = AF_INET;

    // assign the service port
    sinRemote.sin_port = htons(atoi(pszRemoteService));
    if (sinRemote.sin_port == 0)
    {
      pSent = getservbyname(pszRemoteService, "tcp");
      if (pSent == NULL)
      {
      m_nLastError = WSAGetLastError();
      nStatus = CL_WINSOCK_WINSOCK_ERROR;
      break;
      }
      sinRemote.sin_port = pSent->s_port;
    }

    // assign the IP address
    sinRemote.sin_addr.s_addr = inet_addr(pszRemoteName);
    if (sinRemote.sin_addr.s_addr == INADDR_NONE)
    {
      pHent = gethostbyname(pszRemoteName);
      if (pHent == NULL)
```

```
      {
       m_nLastError = WSAGetLastError();
       nStatus = CL_WINSOCK_WINSOCK_ERROR;
       break;
      }
      sinRemote.sin_addr.s_addr = *(u_long *)pHent->h_addr;
    }

    // call the version of Connect() that takes an IP address
    // structure
    return Connect(&sinRemote);
  }

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_StreamSock::Connect()
//
// This version of the Connect() function takes a pointer to an IP
// address structure to connect to.
//
////////////////////////////////////////////////////////////////////

int CL_StreamSock::Connect(LPSOCKADDR_IN psinRemote)
{
  int nStatus = CL_WINSOCK_NOERROR;

  while (TRUE)
  {
    // only clients are allowed to call connect()
    if (m_bServer)
    {
      nStatus = CL_WINSOCK_PROGRAMMING_ERROR;
      break;
    }

    // copy the IP address of the remote connecting server
    memcpy(&m_sinRemote, psinRemote, sizeof(SOCKADDR_IN));

    // attempt the asynchronous connection
    if (connect(m_s,(LPSOCKADDR)&m_sinRemote, sizeof(SOCKADDR_IN)) ==
                  SOCKET_ERROR)
    {
      m_nLastError = WSAGetLastError();
      if (m_nLastError == WSAEWOULDBLOCK)
        m_nLastError = NoErr;
      else
        nStatus = CL_WINSOCK_WINSOCK_ERROR;
      break;
    }
    break;
  }

  return nStatus;
```

```
}
////////////////////////////////////////////////////////////////////
// CL_StreamSock::Accept()
//
// To accept a connection request from a client.
//
////////////////////////////////////////////////////////////////////

int CL_StreamSock::Accept(CL_StreamSock *pStreamSocket)
{
  int nStatus = CL_WINSOCK_NOERROR;

  while (TRUE)
  {
    if (pStreamSocket == NULL)
    {
      ASSERT(0);
      nStatus = CL_WINSOCK_PROGRAMMING_ERROR;
      break;
    }

    // only servers should call Accept()
    if (!m_bServer)
    {
      nStatus = CL_WINSOCK_PROGRAMMING_ERROR;
      break;
    }

    // Check if the socket is not already created.
    if (pStreamSocket->m_s != INVALID_SOCKET)
       return CL_WINSOCK_PROGRAMMING_ERROR;

    pStreamSocket->InitVars();

    // create the hidden window
    RECT rect;
    rect.left = 0;
    rect.top = 0;
    rect.right = 50;
    rect.bottom = 50;
    if (pStreamSocket->Create(NULL, NULL, WS_OVERLAPPEDWINDOW, rect,
                        pStreamSocket->m_pParentWnd, 0) == 0)
    {
      nStatus = CL_WINSOCK_WINDOWS_ERROR;
      break;
    }

    // accept the client connection
    pStreamSocket->m_s = accept(m_s, NULL, NULL);
    if (pStreamSocket->m_s == INVALID_SOCKET)
    {
      m_nLastError = WSAGetLastError();
      nStatus = CL_WINSOCK_WINSOCK_ERROR;
      pStreamSocket->DestroyWindow();
      break;
    }
```

```
    // start the asynchronous event notification
    long lEvent;
    lEvent = FD_READ | FD_WRITE | FD_CONNECT | FD_CLOSE;
    if (WSAAsyncSelect(pStreamSocket->m_s, pStreamSocket->m_hWnd,
        CL_WINSOCK_EVENT_NOTIFICATION, lEvent) == SOCKET_ERROR)
    {
      m_nLastError = WSAGetLastError();
      nStatus = CL_WINSOCK_WINSOCK_ERROR;
      closesocket(pStreamSocket->m_s);
      pStreamSocket->DestroySocket();
      break;
    }

    break;
  }

  // if anything failed in this function, reset the socket variables
  if (nStatus == CL_WINSOCK_WINSOCK_ERROR)
      pStreamSocket->InitVars(FALSE);
  else if (nStatus == CL_WINSOCK_NOERROR)
  {
      // notify the parent if the connection was accepted
      // successfully
      pStreamSocket->m_pParentWnd->PostMessage(pStreamSocket->m_uMsg,
                                CL_WINSOCK_YOU_ARE_CONNECTED);
  }

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_StreamSock::Write()
//
// Write data to the stream socket.
//
////////////////////////////////////////////////////////////////////

int CL_StreamSock::Write(int nLen, LPVOID pData)
{
  int nStatus = CL_WINSOCK_NOERROR;

  while (TRUE)
  {
    // dynamically allocate a structure to hold the data pointer
    // and the data's length
    LPSTREAMDATA pStreamData = new STREAMDATA;
    if (pStreamData == NULL)
    {
      nStatus = CL_WINSOCK_WINDOWS_ERROR;
      break;
    }
    pStreamData->pData = pData;
    pStreamData->nLen = nLen;
```

```
    // add the data to the list
    TRY
    {
      m_listWrite.AddTail(pStreamData);
    }

    CATCH (CMemoryException, e)
    {
      delete pStreamData;
      nStatus = CL_WINSOCK_WINDOWS_ERROR;
      break;
    }
    END_CATCH

    // trigger the FD_WRITE handler to try to send
    PostMessage(CL_WINSOCK_EVENT_NOTIFICATION, m_s,
                WSAMAKESELECTREPLY(FD_WRITE, 0));
    break;
  }

  return nStatus;
}


////////////////////////////////////////////////////////////////////
// CL_StreamSock::Read()
//
// Read data that has been received by the stream socket.
//
////////////////////////////////////////////////////////////////////

LPVOID CL_StreamSock::Read(LPINT pnLen)
{

  LPVOID pData = NULL;

  // check to see if there is data to receive
  if (!m_listRead.IsEmpty())
  {

    // remove the stream data from the list
    LPSTREAMDATA pStreamData = (LPSTREAMDATA)m_listRead.RemoveHead();
    pData = pStreamData->pData;
    *pnLen = pStreamData->nLen;
    delete pStreamData;
  }

  return pData;
}


// message map
BEGIN_MESSAGE_MAP(CL_StreamSock, CWnd)
  ON_MESSAGE(CL_WINSOCK_EVENT_NOTIFICATION, OnWinSockEvent)
END_MESSAGE_MAP()
```

```
////////////////////////////////////////////////////////////////////////
// CL_StreamSock::OnWinSockEvent()
//
// Event handler: Called when there is an asynchronous event on the
// socket.
//
////////////////////////////////////////////////////////////////////////

LONG CL_StreamSock::OnWinSockEvent(WPARAM wParam, LPARAM lParam)
{
  // check for an error
  if (WSAGETSELECTERROR(lParam) != 0)
    return 0L;

  // act upon the notified event
  switch (WSAGETSELECTEVENT(lParam))
  {
    case FD_ACCEPT:
      // inform the parent window that a client would like to connect
      // to the server socket
      m_pParentWnd->PostMessage(m_uMsg,
                    CL_WINSOCK_READY_TO_ACCEPT_CONNECTION);
      break;

    case FD_CONNECT:
      // inform the parent window that the socket has connected
      m_pParentWnd->PostMessage(m_uMsg,
                    CL_WINSOCK_YOU_ARE_CONNECTED);
      break;

    case FD_READ:
      return HandleRead(wParam, lParam);
      break;

    case FD_WRITE:
      return HandleWrite(wParam, lParam);
      break;

    case FD_CLOSE:
      // check for more data queued on the socket
      if (HandleRead(wParam, lParam))
      {
      PostMessage(CL_WINSOCK_EVENT_NOTIFICATION, wParam, lParam);
      break;
      }
      // inform the parent window that the socket is closed
      m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_LOST_CONNECTION);
      break;

    default:
      // exception handling
      ASSERT(0);
      break;
  }
  return 0L;
```

```
}
////////////////////////////////////////////////////////////////////
// CL_StreamSock::HandleRead()
//
// Called when there is an asynchronous read event on the socket.
//
////////////////////////////////////////////////////////////////////

LONG CL_StreamSock::HandleRead(WPARAM wParam, LPARAM lParam)
{
  while (TRUE)
  {
    // allocate memory for incoming data
    LPVOID pData = malloc(READ_BUF_LEN);
    LPSTREAMDATA pStreamData = new STREAMDATA;

    if ((pData == NULL) || (pStreamData == NULL))
    {
      // free any memory that was allocated
      if (pData != NULL)
      free(pData);
      pData = NULL;

      if (pStreamData != NULL)
      delete pStreamData;
      pStreamData = NULL;

      // inform the parent that a possible data read failed
      m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_ERROR_READING);

      PostMessage(CL_WINSOCK_EVENT_NOTIFICATION, m_s,
                  WSAMAKESELECTREPLY(FD_READ, 0));

      break;
    }

    // receive data
    int nBytesRead = recv(m_s, (LPSTR)pData, READ_BUF_LEN, 0);
    if (nBytesRead == SOCKET_ERROR)
    {
      // free memory buffer for incoming data
      free(pData);
      pData = NULL;
      delete pStreamData;
      pStreamData = NULL;

      m_nLastError = WSAGetLastError();
      if (m_nLastError == WSAEWOULDBLOCK)
      m_nLastError = NoErr;
      else
      // inform the parent that a data read failed
      m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_ERROR_READING);

      break;
    }
```

```
    // make sure some data was read
    if (nBytesRead == 0)
    {
      // free memory for incoming data
      free(pData);
      pData = NULL;
      delete pStreamData;
      pStreamData = NULL;

      break;
    }

    // add the data to the list
    pStreamData->pData = pData;
    pStreamData->nLen = nBytesRead;
    TRY
    {
      m_listRead.AddTail(pStreamData);
    }
    CATCH (CMemoryException, e)
    {
      free(pData);
      pData = NULL;
      delete pStreamData;
      pStreamData = NULL;

      // inform the parent that a data read failed
      m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_ERROR_READING);
      break;
    }
    END_CATCH

    // inform the parent that data has been read
    m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_DONE_READING,
                        (LPARAM)m_listRead.GetCount());

    // return 1 if there is still remaining data
    return 1L;

    break;
  }

  return 0L;
}


////////////////////////////////////////////////////////////////////
// CL_StreamSock::HandleWrite()
//
// Called when there is an asynchronous write event on the socket.
//
////////////////////////////////////////////////////////////////////

LONG CL_StreamSock::HandleWrite(WPARAM wParam, LPARAM lParam)
{
```

```
int nLen;
LPVOID pData;
LPSTREAMDATA pStreamData;
static LPVOID pDataRemaining = NULL;
static int nLenRemaining = 0;

while (TRUE)
{
  // check if there is any data to send
  if (m_listWrite.IsEmpty())
    break;

  // if not in the middle of another buffer send,
  // get data and data length from the write queue
  pStreamData = (LPSTREAMDATA)m_listWrite.GetHead();
  pData = pStreamData->pData;
  nLen = pStreamData->nLen;
  if (pDataRemaining == NULL)
  {
     pDataRemaining = pData;
     nLenRemaining = nLen;
  }

  // sending the data
  BOOL bRemove = FALSE;
  int nBytesSent = send(m_s, (LPCSTR)pDataRemaining,
                        nLenRemaining, 0);
  if (nBytesSent == SOCKET_ERROR)
  {
    m_nLastError = WSAGetLastError();
    if (m_nLastError == WSAEWOULDBLOCK)
        m_nLastError = NoErr;
    else
    {
       bRemove = TRUE;
       m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_WRITE_ERR,
                                 (LPARAM)pData);
    }
  }
  else
  {
    // if data was sent, check if all the bytes were sent
    if (nBytesSent == nLenRemaining)
    {
    bRemove = TRUE;
    m_pParentWnd->PostMessage(m_uMsg, CL_WINSOCK_DONE_WRITING,
                              (LPARAM)pData);
    }
    else
    {
    // the complete buffer was not sent so reset these values
    pDataRemaining = (LPVOID)((LPCSTR)pDataRemaining + nBytesSent);
    nLenRemaining = nLenRemaining - nBytesSent;
    }
  }
```

```
    // if the data was completely sent or an error has actually
    // occurred, clean up remaining data from the queue
    if (bRemove)
    {
      delete pStreamData;
      m_listWrite.RemoveHead();
      pDataRemaining = NULL;
      nLenRemaining = 0;
    }

    // if there is more data to send, trigger this FD_WRITE handler
    if ( !m_listWrite.IsEmpty() )
       PostMessage(CL_WINSOCK_EVENT_NOTIFICATION, m_s,
                   WSAMAKESELECTREPLY(FD_WRITE, 0));

    break;
  }

  return 0L;
}


//////////////////////////////////////////////////////////////////////
// CL_StreamSock::GetPeerName()
//
// To copy the IP address of the other end (peer) of the socket
connection into
// the given pointer. Useful for server's to use after an Accept().
//
//////////////////////////////////////////////////////////////////////

int CL_StreamSock::GetPeerName(LPSOCKADDR_IN psinRemote)
{
  int nStatus = CL_WINSOCK_NOERROR;
  int nLen = sizeof(SOCKADDR_IN);

  // check if the listening socket is not calling this function
  if (m_bServer)
      nStatus = CL_WINSOCK_PROGRAMMING_ERROR;
  else if (getpeername(m_s, (LPSOCKADDR)psinRemote, &nLen) ==
           SOCKET_ERROR)
  {
       m_nLastError = WSAGetLastError();
       nStatus = CL_WINSOCK_WINSOCK_ERROR;
  }

  return nStatus;
}
```

Chung, Edward, Chi-Fai. Ph.D. June, 1996

Electrical and Computer Engineering

Quality of Service Analysis for Distributed Multimedia Systems in a Local Area Networking Environment (148 pp.)

Director of Dissertation: Dr. Mehmet Celenk

The stringent timing requirements imposed by distributed multimedia applications have raised questions about the adequacy of continuous media support in the current commercial operating systems. The main objective of this research is to study the requirements, also known as Quality of Service (QOS), of multimedia applications and develop a QOS management scheme to support an efficient multimedia networking environment. An integrated QOS management architecture is proposed to maintain synchronization among different continuous media objects.

The primary goal of this research is to present a set of key application QOS parameters and map these requirements through all the layers of our proposed integrated QOS management framework. Emphasis is placed on four performance criteria for continuous media communication: Throughput, transmission delay, delay variations, and error rates.

End-to-end QOS guarantees are ensured by dynamic QOS control that is orchestrated by a protocol entity called the QOS negotiation agent. The QOS