ATOM: A distributed system for video retrieval via ATM networks

---

Prepared by:

Matthew John Roux

Prepared for:

Mr. M.J. Ventura

Department of Electrical Engineering

University of Cape Town.

This dissertation is submitted to the University of Cape Town in fulfillment of the academic requirements for the Degree of Master of Science in Electrical Engineering.

Cape Town, 30th September 1999

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Declaration

I declare that this dissertation is my own work. Where collaboration with other parties has occurred, or material generated by other researchers is included, the parties and/or material are indicated in the acknowledgements or references as appropriate.

The dissertation is being submitted for the Degree of Master of Science in Electrical Engineering at the University of Cape Town. It has not been submitted before for any degree or examination at any other University.

Signed by candidate

Signature Removed

Matthew John Roux

30TH day of SEPTEMBER 1999

Acknowledgements

I wish to express my sincere gratitude to the following people, for direct and indirect assistance in the preparation of this dissertation:

Mr. M.J. Ventura, my supervisor in the Department of Electrical Engineering at the University of Cape Town, who provided guidance and academic insight.

Patrick Vine, my former partner on this project, for insight, hard work and genuine interest in the subject.

My colleagues in the Communications Research Group (CRG), University of Cape Town, for constructive advice on the designs proposed herein.

My wife, Jackie, for her constant and unfailing support.

Synopsis

The convergence of high speed networks, powerful personal computer processors and improved storage technology has led to the development of video-on-demand services to the desktop that provide interactive controls and deliver Client-selected video information on a Client-specified schedule. This dissertation presents the design of a video-on-demand system for Asynchronous Transfer Mode (ATM) networks, incorporating an optimised topology for the nodes in the system and an architecture for Quality of Service (QoS). The system is called ATOM which stands for Asynchronous Transfer Mode Objects. Real-time video playback over a network consumes large bandwidth and requires strict bounds on delay and error in order to satisfy the visual and auditory needs of the user. Streamed video is a fundamentally different type of traffic to conventional IP (Internet Protocol) data since files are viewed in real-time, not downloaded and then viewed. This streaming data must arrive at the Client decoder when needed or it loses its interactive value. Characteristics of multimedia data are investigated including the use of compression to reduce the excessive bit rates and storage requirements of digital video. The suitability of MPEG-1 for video-on-demand is presented.

Having considered the bandwidth, delay and error requirements of real-time video, the next step in designing the system is to evaluate current models of video-on-demand. The distributed nature of four such models is considered, focusing on how Clients discover Servers and locate videos. This evaluation eliminates a centralized approach in which Servers have no logical or physical connection to any other Servers in the network and also introduces the concept of a selection strategy to find alternative Servers when Servers are fully loaded. During this investigation, it becomes clear that another entity (called a Broker) could provide a central repository for Server information. Clients have logical access to all videos on every Server simply by connecting to a Broker.

The ATOM Model for distributed video-on-demand is then presented by way of a diagram of the topology showing the interconnection of Servers, Brokers and Clients; a description of each node in the system; a list of the connectivity rules; a description of the protocol; a description of the Server selection strategy and the protocol if a Broker fails. A sample network is provided with an example of video selection and design issues are raised and solved including how nodes discover each other, a justification for using a mesh topology for the Broker connections, how Connection Admission Control (CAC) is achieved, how customer billing is achieved and how information security is maintained. A calculation of the number of Servers and Brokers required to

service a particular number of Clients is presented. The advantages of ATOM are described. The underlying distributed connectivity is abstracted away from the Client. Redundant Server/Broker connections are eliminated and the total number of connections in the system are minimized by the rule stating that Clients and Servers may only connect to one Broker at a time. This reduces the total number of Switched Virtual Circuits (SVCs) which are a performance hindrance in ATM. ATOM can be easily scaled by adding more Servers which increases the total system capacity in terms of storage and bandwidth.

In order to transport video satisfactorily, a guaranteed end-to-end Quality of Service architecture must be in place. The design methodology for such an architecture is investigated starting with a review of current QoS architectures in the literature which highlights important definitions including a flow, a service contract and flow management. A flow is a single media source which traverses resource modules between Server and Client. The concept of a flow is important because it enables the identification of the areas requiring consideration when designing a QoS architecture. It is shown that ATOM adheres to the principles motivating the design of a QoS architecture, namely the Integration, Separation and Transparency principles. The issue of mapping human requirements to network QoS parameters is investigated and the action of a QoS framework is introduced, including several possible causes of QoS degradation. The design of the ATOM Quality of Service Architecture (AQOSA) is then presented. AQOSA consists of 11 modules which interact to provide end-to-end QoS guarantees for each stream. Several important results arise from the design. It is shown that intelligent choice of stored videos in respect of peak bandwidth can improve overall system capacity. The concept of disk striping over a disk array is introduced and a Data Placement Strategy is designed which eliminates disk hot spots (i.e. overuse of some disks whilst others lie idle.) A novel parameter (the B-P Ratio) is presented which can be used by the Server to predict future bursts from each video stream. The use of Traffic Shaping to decrease the load on the network from each stream is presented.

Having investigated four algorithms for rewind and fast-forward in the literature, a rewind and fast-forward algorithm is presented. The method produces a significant decrease in bandwidth, and the resultant stream is very constant, reducing the chance that the stream will add to network congestion. The C++ classes of the Server, Broker and Client are described emphasizing the interaction between classes. The use of ATOM in the Virtual Private Network and the multimedia teaching laboratory is considered.

Conclusions and recommendations for future work are presented. It is concluded that digital video applications require high bandwidth, low error, low delay networks; a video-on-demand system to support large Client

volumes must be distributed, not centralized; control and operation (transport) must be separated; the number of ATM Switched Virtual Circuits (SVCs) must be minimized; the increased connections caused by the Broker mesh is justified by the distributed information gain; a Quality of Service solution must address end-to-end issues. It is recommended that a web front-end for Brokers be developed; the system be tested in a wide area ATM network; the Broker protocol be tested by forcing failure of a Broker and that a proprietary file format for disk striping be implemented.

Table of Contents

List of Illustrations

Tables

Equations

Chapter 1 Introduction

1.1 Background to video-on-demand and Asynchronous Transfer Mode (ATM)

Multimedia applications integrating text, graphics, audio and video are radically changing the way people communicate. Examples of multimedia communication applications are video-on-demand (VoD), video-conferencing, video telephony and email incorporating video and audio. Many commercial implementations are appearing. NTT Software Corporation's *Interspace* is an intranet/internet application allowing hundreds of users to interact in a 3D environment using text chat as well as audio and video communications [1]. Microsoft's *NetShow Theater Server* streams full-screen MPEG video to PC Clients using a distributed architecture [2]. Interactive video delivery services represent a fundamental change to the television service. Broadcast television consists of many stations transmitting in parallel, with the user having to choose one channel at the time specified for the transmission. Video-on-demand makes all programming available to users at any time and allows users the flexibility to select and receive video data in real time over a high-speed network. In the computer communications environment, video-on-demand can be defined as *interactive multimedia in which users can view video streams (including audio) and control the playback using fast-forward, rewind, pause and random jump*. Random jump (or dynamic panning) refers to the ability of the user to jump to any location in the movie using a slider toolbar. From the Client's perspective it is important that any control operation be executed with no service degradation, regardless of any increase in users on the Server. There are many applications for a video-on-demand system:

- **Entertainment** - The most obvious application is commercial entertainment in the form of movies. These Servers could stream video to the home, hotel rooms, cruise ship rooms, hospital rooms and in-flight displays. Many hotels in South Africa now provide crude video-on-demand operations via a television in the room.
- **Education** - Students can watch videos of lectures and documentaries. The ability to control the pace of learning is an obvious advantage as is the flexibility with regard to time and location. A good example can be found at the University of Texas at Arlington. A Centre for Distance Education [3] was founded in 1997 with the goal of "desktop training" or "desktop education", the delivery of educational material to a computer at home or work. South Africa has well-documented educational problems especially in rural

areas, consequently the value of a video-on-demand application to long-distance education is obvious. Video Servers containing syllabus-specific videos could be accessed anytime of the day or night in special multimedia centres.

- **Technical and Medical Training** - Video is the perfect medium for teaching complex technical, assembly and medical procedures. If a system offers interaction and on-line testing, the learning process is enhanced.
- **Marketing** - Companies can design advertisement videos to supercede traditional forms of advertising. These videos could be downloaded on webpages or included in sales kiosk systems.
- **News, Weather and Documentaries** - Users can view local and international news and documentary video clips as well as local weather forecasts. The option exists for the user to see more detail on selected stories.

Video-on-demand services are classified by the degree of interaction they allow the user. The bottom end of the scale is Broadcast (No-VoD) services similar to television in which the user has no interaction with the system and must start watching at the time specified by the Server. The other levels of interaction are:

- **quasi video-on-demand (Q-VoD)** - Users are grouped based on similar interests. For example, by joining a Sports Group the user receives sports broadcasts. This implies a certain amount of control over content. Users can perform simple control activities by switching between interest groups. But the user still has no control over the delivery of the video stream.
- **near video-on-demand (N-VoD)** - Controls like fast-forward and rewind are simulated by transitions in discrete time intervals e.g. 5 minutes. This can be achieved by having multiple channels with the same programming staggered in time. Clearly this is a highly inefficient and storage intensive solution to on-demand programming.
- **true video-on-demand (T-VoD)** - The user has complete control over the session presentation. All VCR controls are available in real-time. The design presented in this dissertation aims to deliver true video-on-demand.

Video-on-demand is typically implemented using the Client/Server paradigm. The following definition exists for Client/Server computing: "Client/Server computing is any application in which the requester of actions or information is on one system and the supplier of action or information on another" [4]. The information that can be stored on a Server covers a broad spectrum from movie entertainment to educational documentaries and news video clippings. Figure 1-1 shows the basic components of a VoD system. The multimedia Server has a video library, a disk array, connection admission control and stream management and disk scheduling. The video

streams enter the public Asynchronous Transfer Mode (ATM) network via a local ATM switch. The Client machine receives the video stream from the public network via an ATM access technology, decodes it and displays it on the screen. The question of access technology arises. The Client could either be in an ATM LAN or could make use of an access technology like ADSL (Asymmetric Digital Subscriber Line), Hybrid Fiber Coax (HFC) or fiber-to-the-curb (FTTC). A further consideration is wireless access.

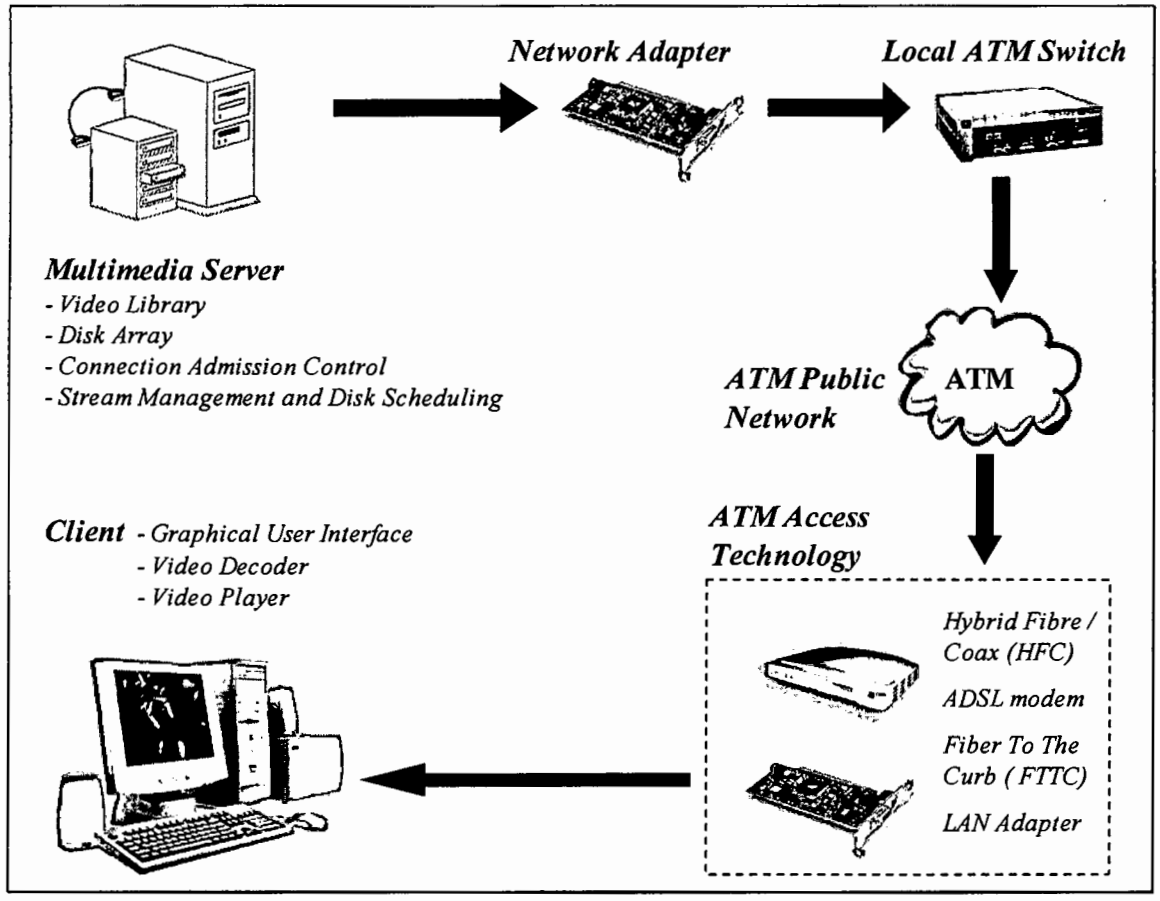

Figure 1-1 The basic components of a VoD system

When multimedia is transmitted over a network, video traffic consumes large quantities of the available bandwidth. Multimedia has strict delay bounds and therefore requires a guaranteed Quality of Service from the network. Compression reduces the peak bandwidth requirements of video significantly but introduces burstiness. Bursty traffic from many streams can quickly result in congestion if peaks coincide. The bursty nature of compressed video requires traffic management in the network to control congestion. Traffic shaping, flow control, congestion control and bandwidth allocation are some of the technical challenges in order to manage the multimedia traffic in a video-on-demand system. A network can be managed more efficiently if the traffic

injected into the network is smoothed. A traffic shaping algorithm can reduce the bursty video as it enters the network. A further technique is to implement flow control so that the source can reduce its bit rate until the network congestion has been eliminated. Intelligent bandwidth allocation algorithms for the video streams can improve network resource utilization.

ATM is a networking technology designed for the high-speed transfer of voice, video and data through public and private networks in a cost effective manner. ATM uses 53 byte cells to transmit data in multiples of OC-1 rates (51.84 Mbits/sec). Popular data transfer rates for ATM are OC-3 (155.52 Mbits/sec) and OC-12 (622.08 Mbits/sec). ATM networks are characterized by their switch-based network architecture. All computers are connected to a switch and communication between pairs of computers is established through the switch. This is in contrast to shared-medium traditional LANs like Ethernet. The aggregate bandwidth of an ATM switch may be several Gbits/sec or more. ATM is expected to serve as the transport mechanism for a wide spectrum of traffic types with varying performance requirements. It is assumed that the reader already has some knowledge of ATM so a detailed description of ATM is omitted, but the topic is covered in [5], [6] and [7]. There are many advantages provided by ATM networks:

- **Connection-oriented** ATM is inherently connection-oriented, allowing for end-to-end QoS setup and cell sequence preservation. This is well-suited for delivery of multimedia.
- **ATM in Internetworks:** ATM has emerged as a technology for integrating LANs and WANs.
- **Performance Guarantees:** Demanding multimedia applications can make use of Quality of Service (QoS) parameters. QoS is an architected feature of ATM.
- **Reduced Downtime:** Due to its low bit error rate, ATM offers high reliability, thereby reducing downtime.
- **Support for legacy protocols:** ATM can support existing network protocols like TCP/IP through Classical IP (CLIP) and Lan Emulation (LANE).
- **Media Independence:** ATM can take full advantage of the large bandwidth of fiber optic media, but can also run over copper using the UTP-5 standard. Current investments in copper cable can therefore maintain their value when networks migrate to full ATM implementations.
- **Integration:** ATM successfully integrates voice, video and data.
- **Scalability:** ATM is capable of accommodating (with appropriate hardware upgrades) the expected growth in end-user bandwidth demand.
- **Intelligent Allocation of Bandwidth:** Bandwidth should be allocated to users intelligently and efficiently. This is possible with ATM since it is part of the original architecture in terms of the statistical multiplexing

of connections. Statistical multiplexing superimposes multiple streams for transport over the same channel and the resulting aggregate traffic exhibits smoother bit rate than the individual streams.

- **Congestion Control:** ATM switches provide traffic management tools designed to reduce congestion e.g. Connection Admission Control, Congestion Notification Messages, Cell Loss Priority bit and Traffic Shaping.
- **ATM is Contractual**: ATM QoS is implemented by parameters contained in an enforceable Traffic Contract. This suits multimedia connections which are best governed by a contract since there are constraints like bandwidth and delay that need to be monitored.

The greatest driving influence behind the multimedia explosion is the dropping of PC component prices. An affordable desktop PC has the computing power, the video card and the RAM to handle multimedia applications. As more people gain interest in multimedia so the multimedia developers improve their applications. Multimedia applications must execute in real time and perform this execution reliably. Multimedia applications have some drawbacks. Multimedia data consumes large storage space even when compressed. This implies the need for dedicated multimedia Servers. Advances in digital encoding and compression [8] have also facilitated the multimedia explosion since they have reduced the raw bandwidth requirements of video to acceptable levels. The parallel growth of the Internet and multimedia have led to applications like streaming video, online shopping and video telephony. Multimedia applications require large databases. Advances in relational database management systems (RDMS) have aided advances in multimedia development. It is clear that multimedia is here to stay and thus deserves well-researched implementations. A video-on-demand system must attract a significant customer base in order to be commercially viable. Such a system must:

- Offer flexible user operation via a well-designed Graphical User Interface (GUI) which could include a measure of customization so that users can define and modify the way services are delivered to them to accommodate individual needs;
- Offer guaranteed Quality of Service;
- Offer sufficient content;
- Compare favourably with stand-alone systems;
- Avoid obsolescence (i.e. it must be future-proof).

1.2 Specifications and Related Work

There are a number of industry specifications and related projects which can provide a foundation for the ATOM Model. Specifications from the ATM Forum, ITU-T, DSM-CC, DAVIC, TINA-C and CORBA are introduced in Section 1.4.1 and related work in the field of video-on-demand and distributed computing is introduced in Section 1.4.2.

1.2.1 Specifications

The ATM Forum is an international non-profit organization formed with the objective of accelerating the use of ATM products and services through a rapid convergence of interoperability specifications. The Forum has defined a specification for video-on-demand. The latest version is Audiovisual Multimedia Services: Video-on-Demand Specification 1.1, March, 1997 [9]. The purpose of this specification is to address the carriage of MPEG-2 bit streams over ATM. The following areas are specified:

- AAL Requirements;
- the encapsulation of MPEG-2 Transport Streams into AAL-5 PDUs;
- the ATM signaling and ATM connection control requirements;
- the traffic characteristics;
- the Quality of Service characteristics;
- reference models for the service.

The following ITU-T specifications refer to video-on-demand:

- *[I.375.2] Recommendation I.375.2 (06/98) - Network capabilities to support multimedia services: Example of multimedia retrieval service class - Video-on-demand service using an ATM based network.* This recommendation specifies network requirements for the video-on-demand service. The network capabilities to support the service are described by means of reference configurations and corresponding network architectures.

- *[H.222.1] Recommendation H.222.1 (03/96) - Multimedia multiplex and synchronization for audiovisual communication in ATM environments.* This recommendation describes the multiplexing and synchronization of multimedia information for audiovisual communications in ATM environments.

DSM-CC (Digital Storage Media - Command and Control) is an ISO/IEC standard [10] developed for the delivery of multimedia broadband services. The standard was developed to aid the creation of open application service protocols. Without such protocols, every service delivered to the home would require its own interface to receive it. Open protocols would allow set-top boxes, PCs or other information appliances to access multiple services from multiple service providers. DSM-CC is transport layer independent. Any application written to use DSM-CC need not care about the underlying transport layer between Server and Client. DSM-CC covers a number of distinct protocol areas: Network Session and Resource Control; Configuration of a Client; Downloading to a Client; VCR-like control of the video stream; Generic Interactive Application Services; Generic Broadcast Application Services. Balabanian et al [11] provide an introductory overview of the different functions covered by the DSM-CC standard. The importance of this standard is the separation of control/management and transport. This idea is maintained in ATOM.

DAVIC (the Digital Audio-Visual Council) is a non-profit association with a membership of over 200 companies from more than 20 countries. It represents all sectors of the audio-visual industry: manufacturing (computer, consumer electronics and telecommunications equipment) and service (broadcasting, telecommunications and CATV), as well as a number of government agencies and research organizations. DAVIC is defining one of the first uses of DSM-CC. DAVIC has adopted DSM-CC in its DAVIC 1.4 Specification [12] where it is used as the protocol for control of multimedia interactive sessions, the resources within the sessions and for service level interactions.

The Telecommunications Information Network Architecture Consortium (TINA-C) is investigating an architecture to provide the basis for a broad spectrum of services, especially broadband and multimedia with the emphasis on services which are independent of the underlying distribution concerns [13]. Once again, this work is important because of this separation of control and transport.

The aim of the ATOM implementation was to adhere to the principles of Object Orientation (OO) when designing the software. Object Orientation is well-documented [14], [5] and the technical principles thereof will not be described in this dissertation. The main advantage of OO programming is the fact that it creates reusable software modules. Once a particular object has been designed and coded, for example an ATM communications class, it can be reused by any program requiring it, for example the Client and Server. This reduces overall development time. Traditional software design models advocate seven phases: requirements specification, analysis, prototyping, evaluation, design, implementation and testing. The first phase is dedicated to the

specification of software requirements and the identification of end users. Taking user requirements into account from the beginning is a crucial part of the application development process and especially so in ATOM, where user QoS for video and audio is crucial. In the analysis phase, abstract models of the real-world objects and their functionality are created. Prototyping offers an opportunity to agree with the user on a user interface design. During the evaluation stage, the prototype should be evaluated by a potential user of the system. Once a solid abstract model of the real-world has been defined, the object-oriented design phase starts finishing with an implementation in the chosen object programming language (Visual C++ in the ATOM case). A testing phase should include the designers and users.

In conclusion, a very important area of these specifications is the concept of separating the control and management of the streams from the actual transport of the streams. ATOM has adopted this concept and maintains separate connections for control and transport data.

1.2.2 Related research and projects

Campbell et al [15] present a Quality of Service Architecture (QoS-A) which offers a framework to specify and implement the required performance properties of multimedia applications over high-performance ATM-based networks. The work is important because it stresses the need for designers to consider the end-to-end requirements of each stream and not just isolated areas of improvement. Andrew Campbell has done an immense amount of work on Multimedia QoS and a list of his publications is available on the World Wide Web at http://www.ctr.columbia.edu/~campbell/andrew/publications/publications.html.

Chan and Wong [16] present a performance study of a distributed video Server system in a fully connected backbone network. They develop an analytical model for a random Server selection strategy which can be used to determine the call blocking probability and the requirements of network bandwidth. The concept of a Server selection strategy is very important since the selection impacts load balancing across the entire distributed system. A list of publications is available on the World Wide Web at http://proj.ee.cityu.edu.hk/networking/ra.html.

Anido and Barnett [17] examine the cost of storage in Video-on-Demand Servers. They show that a distributed approach to Video-on-Demand incurs no greater storage cost than a centralized approach and in fact offers

advantages in terms of bandwidth requirements and service quality. This research strengthens the design decision taken in ATOM to include distributed Servers and Brokers.

The Stony Brook Video Server Project [18] is a distributed video Server application that provides indexing, searching and video streaming in a convenient way to Clients over the network. Developed at the State University of New York, the SBVS allows real-time video to co-exist with non-real-time traffic on an Ethernet LAN, using a real-time Ethernet protocol called RETHER.

Tsang et al [19] present end-to-end performance of digital coded video (MPEG-1, MPEG-2) over a local ATM network. They discuss jitter and frame loss as a function of network load and show that traffic smoothing decreases frame loss significantly while maintaining acceptable jitter and loss bounds. The concept of traffic smoothing is very important and is included on each ATOM Server at the ingress to the network.
The ATOM Source Traffic Model defines the source as variable bit rate MPEG-1 video. Krunz and Tripathi [20] provide a comprehensive model for VBR MPEG-1 traffic which is sufficiently accurate in predicting the queuing performance for real video streams.

Rose [21] presents measurements for different categories of MPEG-1 videos. He shows that typical television sequences like sports, news and music videos encode to MPEG-1 sequences with a high peak bit rate and a high peak-to-mean ratio compared to encoded cinema movie. These results are used in the development of the B-P Ratio in Chapter 4.

Microsoft's NetShow Theater Server streams full-screen MPEG video to PC Clients using a distributed architecture [2]. The distributed architecture consists of a single title Server and one or more content Servers connected by a high-speed network in a fault-tolerant, scalable manner. The emphasis on fault tolerance and scalability is emulated in the design of ATOM.

1.3 Dissertation Objective and Development Methodology

This dissertation presents the design of a distributed VoD system specifically targeted for Asynchronous Transfer Mode (ATM) networks, incorporating an optimised topology and the crucial aspect of Quality of Service (QoS) to provide real-time video playback with strict bounds on delay and error. The system is to be distributed in the local and/or wide area network and must minimize the total number of ATM connections

whilst maximising the Client's chance of finding a desired video. The system is called ATOM, which stands for Asynchronous Transfer Mode Objects, emphazising the distributed nature of the system.

The first step in defining the problem was to investigate the specific end-to-end needs of a digital video stream. This research phase revealed that each digital video has massive storage and bandwidth requirements as well as strict bounds on delay and error. The large storage requirement means that dedicated Servers with large disk capacity are required to host the hundreds or thousands of videos in such a system. This dissertation will show that a Data Placement Strategy on these disks can improve disk access and facilitate interactive controls (like Fast-forward and Rewind). The large bandwidth requirements eliminated legacy networking protocols (e.g. Ethernet) as a transport mechanism and indicated that a protocol supporting Quality of Service agreements and large bandwidth was required. It is shown that ATM is a suitable transport mechanism.

The next step involved a thorough investigation into existing video-on-demand systems with an emphasis on the distributed nature of such systems to see how Clients locate Servers and search for videos. In the video-on-demand context, the distributed environment problem involves intelligent placement of videos on Servers and communication between these video Servers to maximize the Client's chance of viewing a desired video. A well-designed distributed topology will reduce the load experienced by each Server by offering the Client alternative video Servers on which to locate a video. During this investigation, it became clear that another entity (called a Broker) could be used to collate information about the distributed system and thus provide a central repository for Server information. Clients can perform a search of all the videos on every Server simply by connecting to a Broker.

Having decided on the introduction of a Broker, the next step was to decide on the topology of the three nodes (Servers, Brokers and Clients) in the network and to create connectivity rules which reduce redundant connections. In ATM, the time taken to setup a Switched Virtual Circuit (SVC) is an important performance hindrance, necessitating the need to minimize connections. A video-on-demand system consists of SVCs (not Permanent Virtual Circuits) since Client/Server connections are relatively short lived (in the order of minutes and hours). Therefore, the connectivity rules state that Brokers must connect to each other and that a Client or Server may only connect to one Broker at a time. The connected Brokers share all information about connected Servers, thus providing a distributed system in which every Client has access to information about every video on every Server even though only a single connection to the Broker has been made. The Brokers form a fully

connected mesh which provides a highly resilient system. It was necessary to justify the extra connections required for this resilience.

Once the Client has located and connected to the Server, it is necessary to consider all the areas on the path from Server to Client that can be optimized to improve the Quality of Service of that stream. A thorough investigation of existing QoS architectures revealed important areas including data placement on Server disks, scheduling, traffic shaping, usage parameter control and user access pattern characterization. The ATOM Quality of Service Architecture (AQOSA) is designed to provide end-to-end Quality of Service guarantees to each stream.

Next, an object-oriented design and implementation of the ATOM Server, Broker and Client nodes was required to test the design. The platform upon which this system runs is Windows NT 4.0. The system is coded in Visual C++ using Winsock2 as the communications interface to an ATM stack. Broadband-ISDN networks using ATM are best suited for providing video-on-demand services [22]. A method of Fast-forward and Rewind was designed in order to provide true video-on-demand for each ATOM stream.

Finally, conclusions about ATOM were drawn and a list of possible extensions and further work presented.

1.4 Plan of Development of the Dissertation

Chapter 2 defines the characteristics of multimedia data and shows that multimedia applications require large bandwidth and timely delivery from the network in order to satisfy the visual and auditory needs of the user. Video and audio represent a fundamentally different type of traffic to conventional IP (Internet Protocol) data since files are not downloaded completely before playback, but blocks of the video and audio data are sent by the Server and consumed by the Client in real time to produce a constant output. This streamed data must arrive at the decoder when needed or it loses its interactive value. This Chapter provides values for acceptable delay, error and cell loss for MPEG-1 video streams. An overview of MPEG-1 compression is presented and the specific delivery needs of video are investigated in order to show that a high bandwidth network is imperative to the provision of multiple streams. Reasons are given why MPEG-1 is the compression technique chosen for the videos stored in an ATOM Server.

Chapter 3 presents the distributed design of ATOM. First, in order to compare ATOM to existing theory, four current models of distributed VoD systems are evaluated. The distributed nature of each of the models is

considered, focusing on how Clients discover Servers and search for videos. The first model is called the One-to-One model in which a Client connects to a single Server which has no connection to any other Servers. This is a centralized approach with no distributed information. The second model, from Chan and Wong, distributes the videos over multiple Servers and includes a Server selection strategy to find alternative Servers when Servers are fully loaded. The third model, from Anido and Barnett, contains a Front End Server which acts as an agent for Clients wanting to find Servers. The problem with their model is that the Front End Servers also stream video from a cache and therefore there is no separation of control and operation. The fourth model, called the NetShow Theater Server from Microsoft, consists of a single title Server and one or more content Servers connected by a high-speed network (switched Ethernet or ATM) in a fault-tolerant, scalable manner. This system only operates over Classical IP and not native ATM. The ATOM Model for distributed VoD is then presented by way of a diagram of the topology, a description of each node, a list of the connectivity rules, a description of the protocol, a description of the Server selection strategy and the protocol if a Broker fails. A sample network is depicted and an example of video selection is presented. Design issues are raised and solved including how nodes discover each other, a justification for using a mesh topology for the Broker connections, how Connection Admission Control (CAC) is achieved, how customer billing is achieved and how information security is maintained. A calculation of the number of Servers and Brokers required to service a particular number of Clients is presented. ATOM provides many advantages. The underlying distributed connectivity is abstracted away from the Client. Redundant Server/Broker connections are eliminated and the total number of connections in the system are minimized by the rule stating that Clients and Servers may only connect to one Broker at a time. This reduces the total number of Switched Virtual Circuits (SVCs) which are a performance hindrance in ATM. ATOM is scalable since adding more Servers easily increases the total system capacity in terms of storage and bandwidth.

The aim of Chapter 4 is to introduce the design methodology for a QoS implementation and to present the design of the ATOM Quality of Service Architecture (AQOSA). The Chapter begins with a review of current QoS architectures in the literature. This review highlights important definitions including a flow, service contract and flow management. A flow is a single media source (e.g. audio, video or data) which traverses resource modules (e.g. CPU, memory, network) at each layer from source media devices, down through the source protocol stack, through the network, up through the receiver protocol stack to the playout device. The concept of a flow is important because it enables the identification of the areas requiring consideration when designing a QoS architecture. Three principles motivating the design of a generalized QoS architecture are defined, namely the Integration, Separation and Transparency principles and these are adhered to in the design of AQOSA. The issue of mapping human requirements to network QoS parameters is investigated and the action of a QoS framework

is introduced. A QoS framework specifies that Clients present desired parameters for a stream in a Service Contract to the network, which can accept or deny the stream based on available resources. Several possible causes of QoS degradation are listed. The design of AQOSA is then presented. AQOSA consists of 11 modules which work in unison to provide end-to-end QoS guarantees for the video-on-demand streams. Several important results arise from the design. First, it is shown that intelligent choice of stored videos in respect of peak bandwidth can improve overall system capacity. Then the concept of Disk Striping over a disk array is introduced and a Data Placement Strategy is designed which eliminates disk hot spots (i.e. overuse of some disks whilst others lie idle.) Next, a novel parameter (the B-P Ratio) is presented which can be used by the Server to predict future bursts from each video stream. Then the use of Traffic Shaping to decrease the load on the network from each stream is presented.

Chapter 5 presents the Software Implementation and considers possible ATOM applications. First, four algorithms for Fast-forward / Rewind are investigated by showing comparisons for a specific MPEG-1 file. Several problems are highlighted. Two of the algorithms, namely Increased Playback Rate and Skip Independent Sequences, cause the load on both the Server and the network to increase. A third algorithm, namely Skip GOPs, does not increase the visual playback during operation and also requires greater bandwidth. A fourth algorithm, namely Alternative File, increases the storage space required on the Server. The ATOM FF / RW method is presented and applied to the same MPEG-1 file. The method only sends I-frames, but averages the I-frame data over the one second period into 30 blocks of data. Apart from a significant decrease in bandwidth, the resultant stream is very constant, reducing the chance that the stream will add to network congestion. The classes of the Server, Broker and Client are then described. For each class, the included Header files, the external objects and the member functions are listed. A description of the class is then given, emphasizing the interaction between classes. Finally, the use of ATOM in the Virtual Private Network and the multimedia teaching laboratory is considered.

Chapter 6 presents conclusions and recommendations for future work. It is concluded that digital video applications require high bandwidth, low error, low delay networks; a video-on-demand system to support large Client volumes must be distributed, not centralized; control and operation (transport) must be separated; the number of ATM Switched Virtual Circuits (SVCs) must be minimized; the increased connections caused by the Broker mesh is justified by the distributed information gain; a Quality of Service solution must address end-to-end issues. It is recommended that a web front-end for Brokers be developed; the system be tested in a wide area

ATM network; the Broker protocol be tested by forcing failure of a Broker and that a proprietary file format for disk striping be implemented.

The literature review is distributed around the dissertation enabling important concepts to be covered as they arise.

A note on calculations:

Whenever a calculation involving bits or bytes is presented in this dissertation, the following convention is adopted:

1 Kilobit = 1024 bits = 1 Kbit
1 Megabit = 1024 * 1024 bits = 1048776 bits = 1Mbit

1 Kbyte = 1024 bytes = 1 KB
1 Megabyte = 1024 * 1024 bytes = 1048776 bytes = 1 MB

Chapter 2 Multimedia Characteristics

2.1 Introduction

Multimedia communication applications integrate video, audio and data (e.g. text and graphics). These applications require large bandwidth and timely delivery from the network in order to satisfy the visual and auditory needs of the user. Video and audio represent a fundamentally different type of traffic to conventional IP (Internet Protocol) data traffic. Instead of a file being downloaded completely before playback, blocks of the video and audio data are sent by the Server and consumed by the Client in real time to produce a constant output. This *streaming data* must arrive when needed or it loses its interactive value. This Chapter sets the stage for a video-on-demand system design by providing an overview of multimedia characteristics, including the use of video compression to reduce the massive storage and bandwidth requirements of digital video. An overview of MPEG-1 compression is presented and the specific delivery needs of video are investigated. The issues raised in this Chapter will show that, in order to deliver multimedia satisfactorily, an end-to-end Quality of Service guarantee is necessary.

2.2 Multimedia Characteristics

2.2.1 Video

Video is presented to the human eye as "a series of frames in which the motion of the scene is reflected in small changes in sequentially displayed frames" [6]. Frames are displayed on the screen at a constant frame rate, such as 30 frames per second. Television signals in the United States have 30 discrete frames per second (60 interlaced fields), while European and South African stations broadcast at 25 frames per second (50 interlaced fields). Most analog cinema movies are both filmed and played back at a rate of 24 frames per second. In order to represent analog video digitally, it is necessary to transform a continuous image field into the discrete domain. The transformation maps the continuous image samples into a finite set of discrete amplitudes that span the intensity range of the particular image. Each quantized amplitude has a unique digital code word, called a pixel. For example, full colour images generally require 24 bits/pixel which is 8 bits/pixel each for the red, blue and green (RGB) component.

The bandwidth requirements of uncompressed digitized video far exceed the available resources for the typical end user. The uncompressed bit rate of digitized video signals can vary anywhere from a few to several hundred megabits per second as evidenced in Table 1. In Table 1, the **Typical Video File** assumes a resolution of 320x240 and a frame rate of 15 frames per second. **CGA** and **SVGA** are computer monitor full-screen display standards whilst **NTSC, PAL** and **SECAM** are video signals used by televisions and VCRs in the United States, Europe and France respectively.

Format	Pixels per line	Line per frame	Pixels per Frame	Bits per pixel	Bits per frame	Frames per second	Mbps
Typical Video File	320	240	76800	24	1 843 200	15	26.36
CGA	640	200	128 000	4	512 000	60	29.29
SVGA	800	600	480 000	8	3 840 000	72	263.67
NTSC	600	485	291 000	24	6 984 000	30	199.81
PAL	580	575	333 500	24	8 004 000	25	190.83
SECAM	580	575	333 500	24	8 004 000	25	190.83

Table 1 Bandwidth requirements for uncompressed digital video

Fortunately, visual applications have inherent redundancies. This has led to video compression mechanisms that can be used to reduce these excessive bit rates. Video compression is a process whereby a collection of algorithms and techniques replace the original pixel-related information with a more compact mathematical description [8]. Decompression is the reverse process of decoding the mathematical descriptions back to pixels for display. Sequences of video frames contain significant repetition. Pixels in neighbouring locations of the same frame will be very similar. This is called spatial (or intraframe) redundancy. Subsequent frames have much common data. This is called temporal (or interframe) redundancy. Compression techniques can be used to eliminate these redundancies, thus reducing the large bit rates needed for video transmission as can be seen in Table 2. Motion JPEG uses spatial (intraframe) compression whilst MPEG-1 and MPEG-2 use spatial and temporal (interframe) compression.

Standard/Format	Data Rate Range	Compression compared to Broadcast quality
Motion JPEG	10-20 Mbps	7-27 times compressed
MPEG-1	1.5 - 4.0 Mbps	100 times compressed
MPEG-2	4 - 60 Mbps	30-100 times compressed

H.261	64 Kbps - 2 Mbps	24 times compressed
DVI (Digital Video Interactive)	1.2 - 1.5 Mbps	160 times compressed
CD-I (Compact Disc Interactive)	1.2 - 1.5 Mbps	100 times compressed

Table 2 Compression standards

Figure 2-1 details the transmission of compressed video over a network. Analog video files are digitized, compressed and stored on the Server. The user requests the transmission of a compressed video over the network. The Client software receives and decompresses the video for playback. This is the basis of video-on-demand.

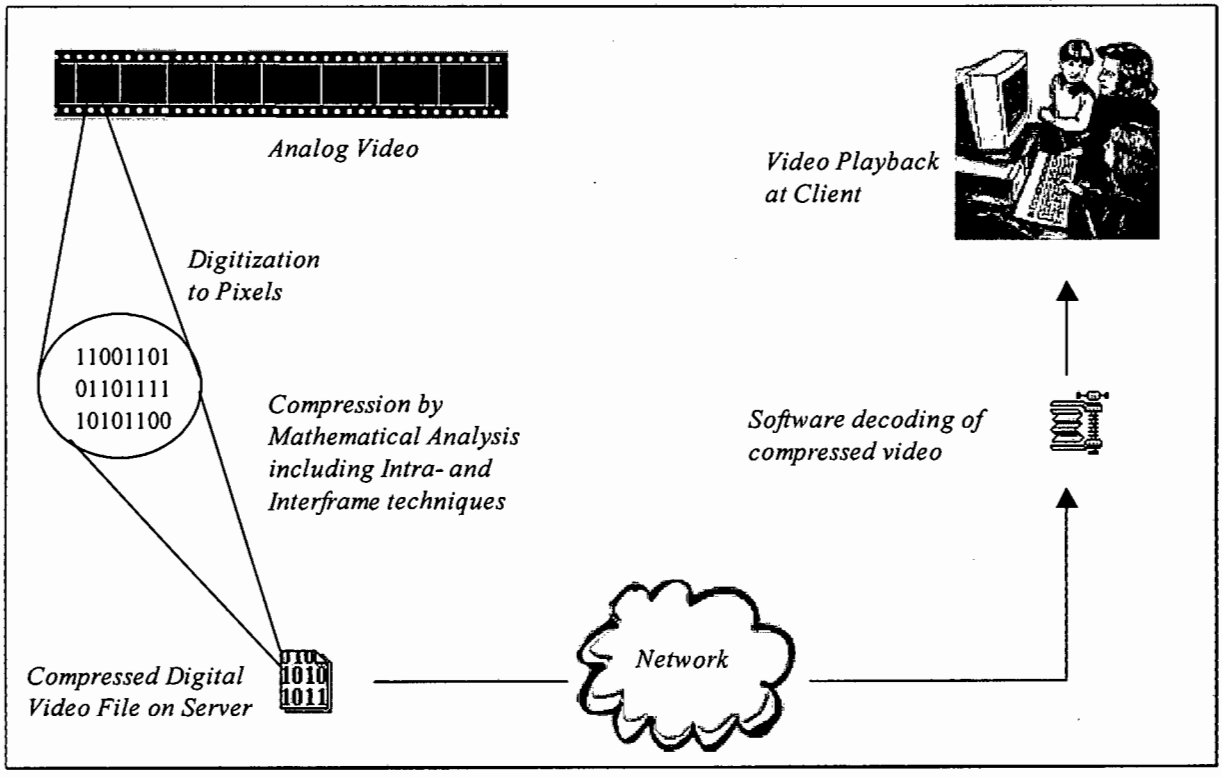

Figure 2-1 Transmission of compressed video over a network

The decompression of the received video can occur in hardware or software. Hardware-assisted decompression typically offers full-screen video at 30 frames per second and requires a separate video decompression card. Software-only decompression uses the host CPU. This offers smaller resolutions and lower frame rates, but with the advantage that no extra hardware need be bought. Decompression in the implementation of ATOM occurs in software. Compression technology can be classified into two groups:

1. Constant Bit Rate (CBR) in which the output rate of the encoder is held constant, producing a video of varying quality.
2. Variable Bit Rate (VBR) in which the output rate of the encoder is variable but the quality of the video is constant.

CBR video provides the advantage that the network knows the bit rate over the full connection time and only Peak Cell Rate monitoring is required. Therefore the Usage Parameter Control (UPC) is simple. VBR video requires more than just a Peak Cell Rate parameter for Connection Admission Control and Usage Parameter Control. Chapter 4 investigates the necessary parameters for Connection Admission Control (CAC) and UPC for VBR video.

2.2.2 Audio

Audio is a continuous stream requiring low bandwidth (from 64 Kbps for a standard telephone quality up to few Mbps for multi-channel Hi-Fi stereo music).

2.2.3 Data

In multimedia, data is defined as anything that is not video or audio. A graphic like a *jpeg* or *gif* or a page of text would fit this definition.

2.3 An overview of MPEG-1

MPEG (Moving Picture Experts Group) is a generic means of compactly representing digital video and audio signals for consumer distribution. MPEG-1 is chosen as the compression technique for the videos stored on the ATOM Servers because of the low bit rate. The MPEG-1 data stream consists of six layers: video sequence, Group of Pictures, frame, slice, macroblock and block. For the purposes of this dissertation, the Group of Pictures (GOP) and frame layers are most important. A video sequence consists of frames arranged in a periodic sequence called a Group of Pictures (GOP). An example of a GOP is **I B B P B B P B B P B B P B B**. There are three types of frame in MPEG-1:

- I-frames (I stands for Intracoded Pictures)
- P-frames (P stands for Predictive Pictures)
- B-frames (B stands for Bid-irectional Pictures)

The relationship between these three frames is depicted in Figure 2-2. I-frames are the least compressed frames since they are self-contained still pictures and can stand alone in the video sequence whereas B-frames are the most compressed, but rely on previous or future I and P-frames. I-frames are compressed in a very similar fashion to the JPEG (Joint Photographic Experts Group) still picture compression standard. P-frames use motion compensated prediction from the previous I- or P-frame in the video sequence. The upper arrows in Figure 2-2 indicate this forward prediction. B-frames use motion compensated prediction by either forward predicting from future I or P-frames, backward predicting from previous I or P-frames, or interpolating between both forward and backward I or P-frames. The lower arrows in Figure 2-2 indicate this bi-directional prediction. I-frames are the largest frames, followed by P and then B-frames in decreasing order of size. Statistical properties of MPEG video traffic and their impact on ATM networks are considered in Chapter 4.

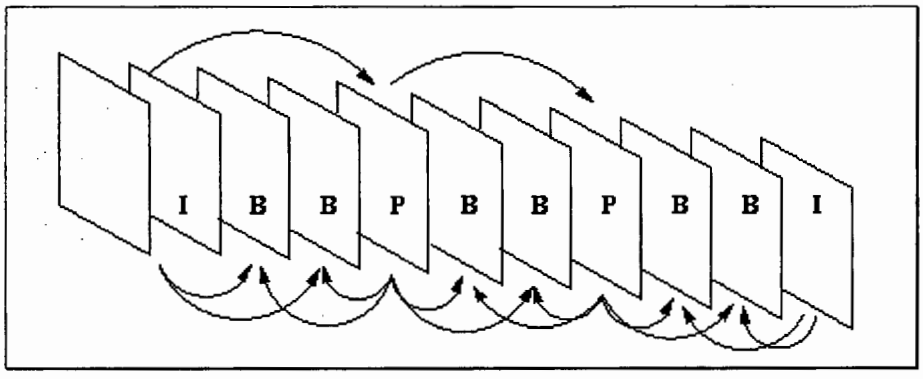

Figure 2-2 Prediction between MPEG-1 Frames (source: [23])

2.4 Requirements for the satisfactory delivery of multimedia

In legacy network protocols like TCP/IP no special emphasis was placed on delay or throughput guarantees. Error-free transmission was the critical aspect for applications like the File Transfer Protocol (FTP). The Internet Protocol (IP) only permits the specification of qualitative QoS parameters like "low" delay, "high" throughput and "high" reliability. The underlying network seldom honours these specifications. Multimedia has different requirements. Multimedia communications need end-to-end performance guarantees because of the time continuous nature of video and audio. This information must arrive without serious delay or it becomes meaningless. Human beings are notoriously intolerant when it comes to lost pictures, distorted audio, blockiness and freeze-framing. Compressed MPEG-1 video has an acceptable end-to-end delay of 100 ms, an acceptable bit

error rate of 10^{-10} and acceptable cell loss ratio of 10^{-12} [24]. These stringent requirements necessitate an end-to-end performance guarantee. The first requirement is that the network protocol be able to satisfy such requirements. The main requirements from the network for the satisfactory delivery of video are as follows:

- **Bandwidth** Even after compression, video files demand large bandwidth (or throughput) from the network. MPEG-1 videos require between 1.5 and 4 Mbps whilst MPEG-2 videos require up to 10 Mbps.

- **Low latency** Latency (or delay) is the time taken for data to flow from source to destination. A bound on this latency is especially necessary for "live" sessions such as long-distance teleconferencing when users at either end must respond to each other without undue delays. Delays are caused on the Server and Client machines due to resource sharing (memory, CPU time and bus access) and in the network due to switch buffers. Latency in an ATM switch ranges from 10 to 60 μs at 7.5 Mbps on a 155 Mbps line [25].

- **Low jitter** Jitter is the variation in latency and is caused by cell buffering in switches. A bound on jitter is required even in one-way applications to avoid frame slips or poor synchronization between audio and video. For most video applications, a jitter (or Cell Delay Variation - CDV) of up to 1 ms is acceptable. The CDV in an ATM switch ranges from 0.92 to 2 μs at 7.5 Mbps on a 155 Mbps line [25].

- **Multicast** It may be feasible that more than one person is watching a particular video at the same time. A multicast algorithm will ensure that the entire video stream is not replicated *n* times for *n* users, but is replicated only at the last possible branch point to each user.

- **Low Error** Either bit error rate or cell loss rate. Bit error is normally caused by physical transmission errors such as noise and interference and is low in modern networks. Fiber optic networks normally operate at error rates of 10^{-9} to 10^{-11}. Overflowing buffers at congested network switches cause cell loss. AAL5 implements a 32-bit Cyclic Redundancy Check (CRC) in its Convergence Sublayer which deals with long bursts of up to 32 bit errors due to physical failures [24].

It is important to note that the compression of digital video removes a large portion of the redundancy present in the original video, thus increasing the impact of cell loss on the quality of service. However, the user may tolerate a certain amount of distortion and video traffic is more tolerant of cell loss and cell errors than audio because viewers do not notice the loss of a few bits and the video decompression can often compensate for such errors. In extreme cases, lost picture frames simply drop out until the next frame arrives one-thirtieth of a second later, in the case of 30-frame-per-second transmission. The implication is that unlike data packets that are lost or erred, video is never retransmitted to avoid mixing uP-frame sequences. In summary, the three important issues for the delivery of video are: Bandwidth, Delay and Error Rate.

Chapter 3 The Design of the Distributed System

3.1 Introduction to distributed computing and communications

"A distributed computing system contains software programs and data resources dispersed across independent computers connected through a communication network."

Richard M. Adler [26]

Computing is considered to be distributed when the software and data that computers work on are spread out over more than one computer, usually over a network. In the video-on-demand context, the distributed environment problem involves intelligent placement of videos on Servers and communication between these video Servers to maximize the Client's chance of viewing a desired video. A well-designed distributed system will reduce the load experienced by each Server by offering the Client alternative video Servers on which to locate a video. Section 3.2 will present four current models of distributed VoD systems with discussions on their advantages and disadvantages. Section 3.3 will present the ATOM Model for distributed video-on-demand. Each node (Server, Client and Broker) is described and the rules and protocol are established. The Server selection strategy is discussed and the protocol if a Broker fails is presented. A sample network is presented. Crucial design issues are solved in Section 3.4. These design issues investigate how nodes discover each other, whether a mesh connection of Brokers is justified, how Connection Admission Control is achieved, how customer billing is achieved and how information security is maintained. A calculation of the number of Servers and Brokers required to service a particular number of Clients is presented in Section 3.4.2. The advantages of the ATOM Model are listed in Section 3.5.

3.2 Four current models of distributed video-on-demand systems

Chapter 1 and Chapter 2 introduced the basic components of a video-on-demand system in which Clients view compressed videos transmitted by content Servers over a network (see Figure 1-1 and Figure 2-1). This description does not explain how the Client locates the Server, whether Servers have knowledge of each other and how the system handles link failures and other errors. The models in Sections 3.2.1 through 3.2.4 show ways for the Client to locate and connect to the Server. Note that there are two general ways of organizing the Client/Server relationship: centralized or distributed. In a centralized model, one video Server provides all

streams to the end user. In a distributed system, numerous video Servers share the user load. Section 3.2.5 summarizes the four models.

3.2.1 Model 1: One-to-One

This is the traditional centralized Client/Server model in which a Client connects to a stand-alone Server (see Figure 3-1). The Server has no connection to any other Servers. If the Client cannot locate a desired video on the Server, it must disconnect and locate another video Server of its own accord. There is no distributed information since Servers are not aware of each other. If a Client cannot locate a desired video on a Server, it cannot ask to be redirected to another Server to locate the video.

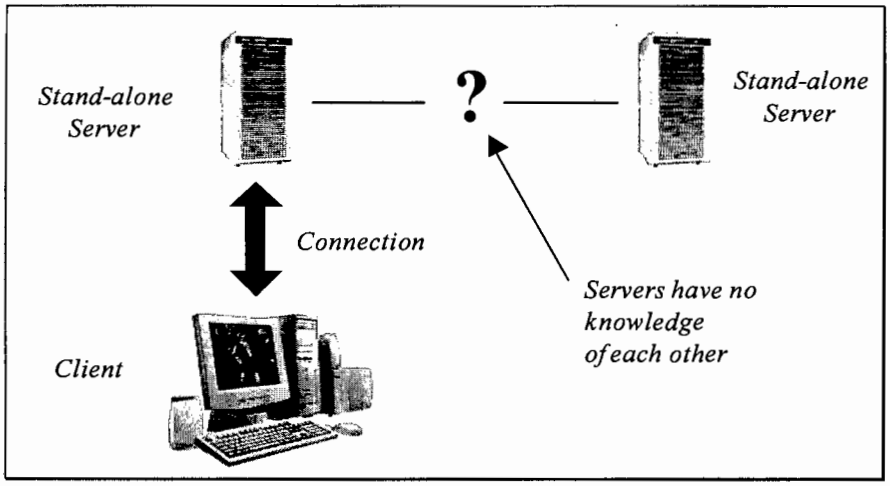

Figure 3-1 One-to-One Model

3.2.2 Model 2: The Chan and Wong central video Server model

Chan and Wong [16] propose a distributed, fully-connected VoD system which consists of two types of video Servers, the local video Server (LVS) and the central video Server (CVS) - see Figure 3-2. The CVS stores all the available videos on CD-ROM or magnetic tape. Each LVS stores only the popular videos, which are downloaded daily or weekly from the CVS. This method reduces the accesses to the CVS in the following way: when a less popular video is requested, the CVS will transmit the requested video to the LVS which will then stream the video to the Client. The request is blocked if the CVS is unable to service it. When a popular video is requested, the nearest LVS will be tried and not the CVS. If it is busy, other LVSs are tried. Clearly, if all LVSs and the CVS are unable to service the video request, it is blocked. The question arises as to how an LVS is

selected. Chan and Wong present several Server Selection Strategies. For all the strategies considered, the user request is assigned to the nearest LVS situated at the local switch first. If this LVS is busy, a remote LVS is tried according to two strategies: Random or Least Loaded. The Random Strategy implies that an LVS is chosen among the available remote LVSs at random. If all are busy, the request is blocked. The Least Loaded Strategy selects the Server with the lightest load. Again, if all are busy, the request is blocked. Accessing remote LVSs implies a greater load on network bandwidth and could lead to congestion. Chan and Wong propose reserving some LVS capacity for local requests only. Chan and Wong do not describe the protocol used for inter-Server communication and fail to describe how the Servers decide amongst themselves which is the Least Loaded.

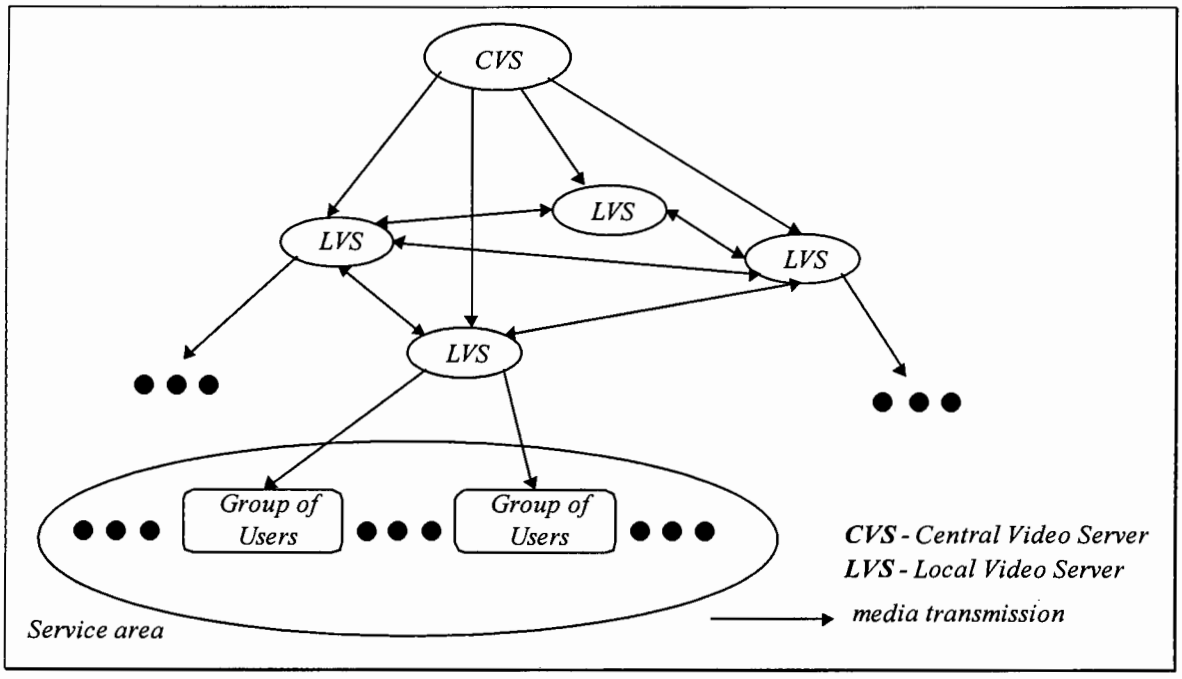

Figure 3-2 A fully-connected Local Video Server (LVS Network) for VoD systems (Source: [16])

3.2.3 Model 3: The front-end Server (FES) model

Anido and Barnett [17] detail a VoD network architecture which contains a front-end Server (FES) with video buffering capabilities at the FES (see Figure 3-3). The system consists of video Servers, set-top boxes (STP) and the FES. A set-top box is able to receive and decode video in place of a computer. The FES is responsible for establishing and managing connections between the STB and the VS. It provides a menu of choices to the user. Anido and Barnett propose the provision of video buffers (or cache) at the FES which will service a certain percentage of user requests. The advantages of this include balancing of network load and reduced delay and

jitter since the source of the video stream is now closer to the set-top box. The obvious drawback is that the FES will then require relatively large amounts of storage. This effectively turns the FES into a VS with added management functionality.

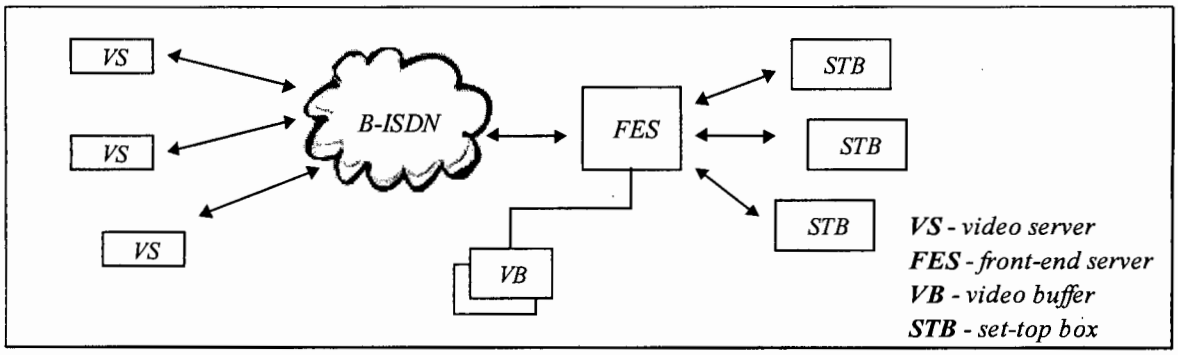

Figure 3-3 Generic VoD Network Architecture - Anido and Barnett [17]

3.2.4 Model 4: Microsoft NetShow Theater Server

Microsoft's NetShow Theater Server streams full-screen MPEG video to PC Clients using a distributed architecture [2] and runs on the Windows NT operating system (see Figure 3-4). The distributed architecture consists of a single title Server and one or more content Servers connected by a high-speed network (switched Ethernet or ATM) in a fault-tolerant, scalable manner. The Title Server receives and forwards Client requests to the content Servers, keeps the system clock and manages system resources. No video data flows through the Title Server. Content Servers retrieve and deliver data to Clients. Each Content Server has 1-20 disk drives that store the data in a proprietary file system. Each video is striped across all disks in the array, balancing user load. Client workstations run Windows Media Player with full VCR functionality. This system is able to provide 60 streams per content Server assuming an ATM OC-3 network card and each stream is 2 Mbps. This means that the disk capacity is limited by the network interface card, since the disk array is able to provide more than 60 simultaneous streams. The NetShow Theater Server has been designed to operate on ATM networks using Classical IP (CLIP) which means that the video is sent in IP packets which are then encapsulated into ATM cells. Native ATM is ignored. If fault-tolerance is enabled, other content Servers and disk drives pick up the load from a failed content Server or disk drive. The user cannot detect a difference in performance, other than a very short interruption in the video (less than 1 second). A mirroring technique is used in which two copies of each video reside in the system, reducing the overall system capacity.

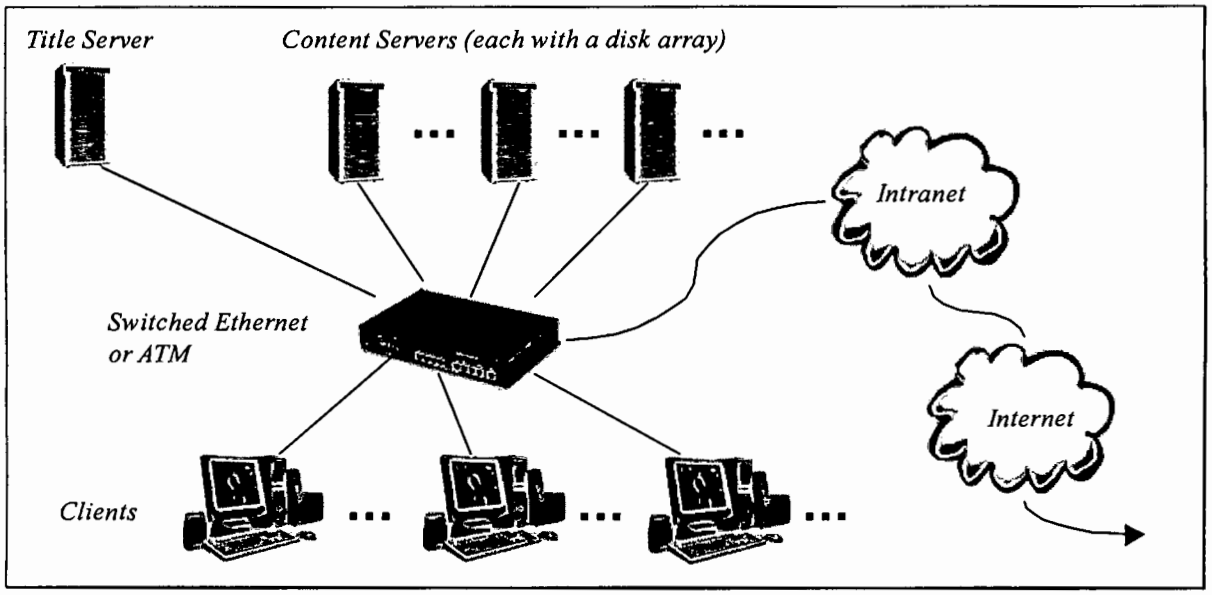

Figure 3-4 Microsoft NetShow Theatre System Architecture

3.2.5 Summary of models

From the above investigation, it is clear that the following possibilities exist: the system can be centralized or distributed. Model 1 is centralized and the others are distributed. Implicit in a distributed environment is that the Servers have some mechanism for co-ordinating content. For example, in Model 2, the local video Servers store the most popular videos. Since the system is distributed, a Client can find more obscure titles on the central video Server. Model 3 contains the intermediate FES which takes connection admission control away from the video Server. Model 4 contains a Title Server which manages Client requests for video titles. Unlike the FES, the Title Server passes no video data, only database-type information.

3.3 The ATOM Model

ATOM is a distributed system consisting of software located on personal computers running Windows NT workstation. The system consists of Servers, Brokers (similar to the FES of the Anido and Barnett model but without the cache, and to the Title Server of the Microsoft NetShow Theater model) and Clients connected according to specific rules of the topology. The system is scalable in terms of the number of users and the number of videos on offer. ATOM is presented in the following way:

- Figure 3-5 depicts the distributed topology of the nodes. Brokers are connected to each other to form a mesh topology and Clients and Servers connect to Brokers to form star topologies. ***Note*:** This topology will exist in an ATM network so the connections between Servers, Clients and Brokers are created through ATM switches as shown in Figure 3-6;
- The nodes (Servers, Clients, Brokers) are described in Section 3.3.1;
- The video database is described in Section 3.3.2;
- The rules of the model are listed and described in Section 3.3.3;
- The Server selection strategy is described in Section 3.3.4;
- The protocol if a Broker fails is covered in Section 3.3.5;
- A sample network configuration built on the topology is presented in Section 3.3.6;
- Design issues are presented in Section 3.4. These design issues cover how nodes discover each other, whether a mesh connection of Brokers is justified, how Connection Admission Control is achieved, how customer billing is achieved and how information security is maintained.
- Advantages of the ATOM Model are presented in Section 3.5.

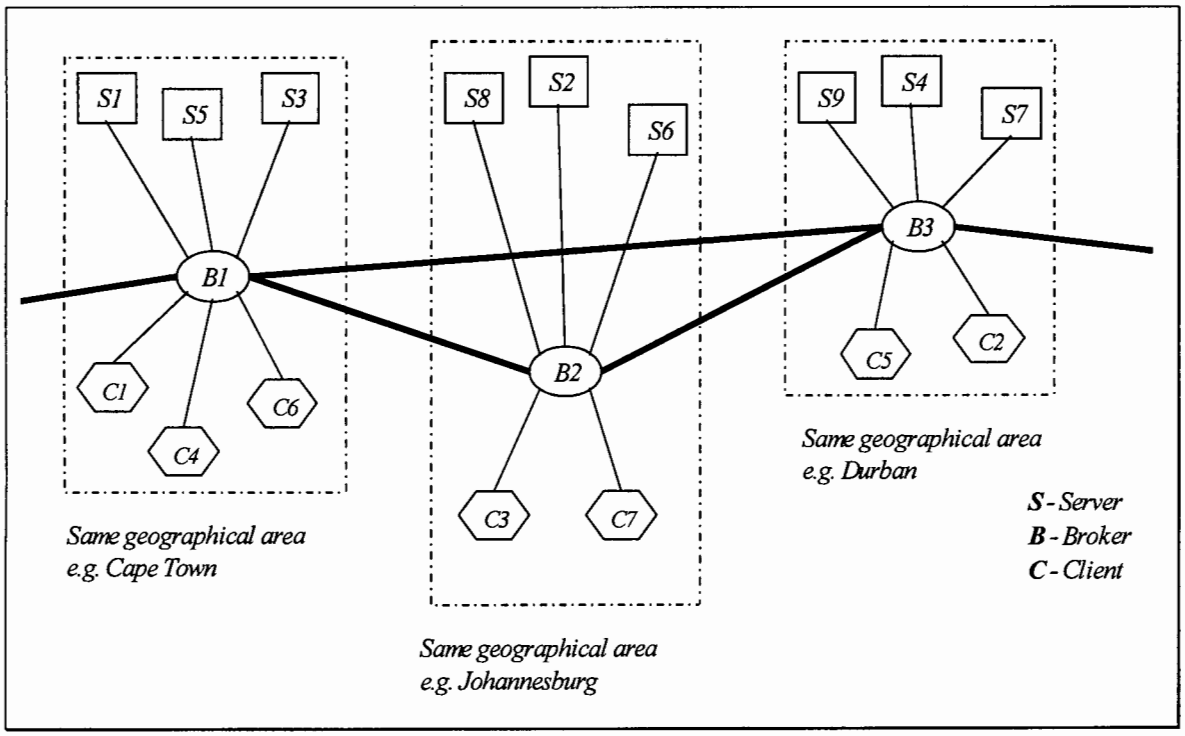

Figure 3-5 Distributed node topology for the ATOM Model

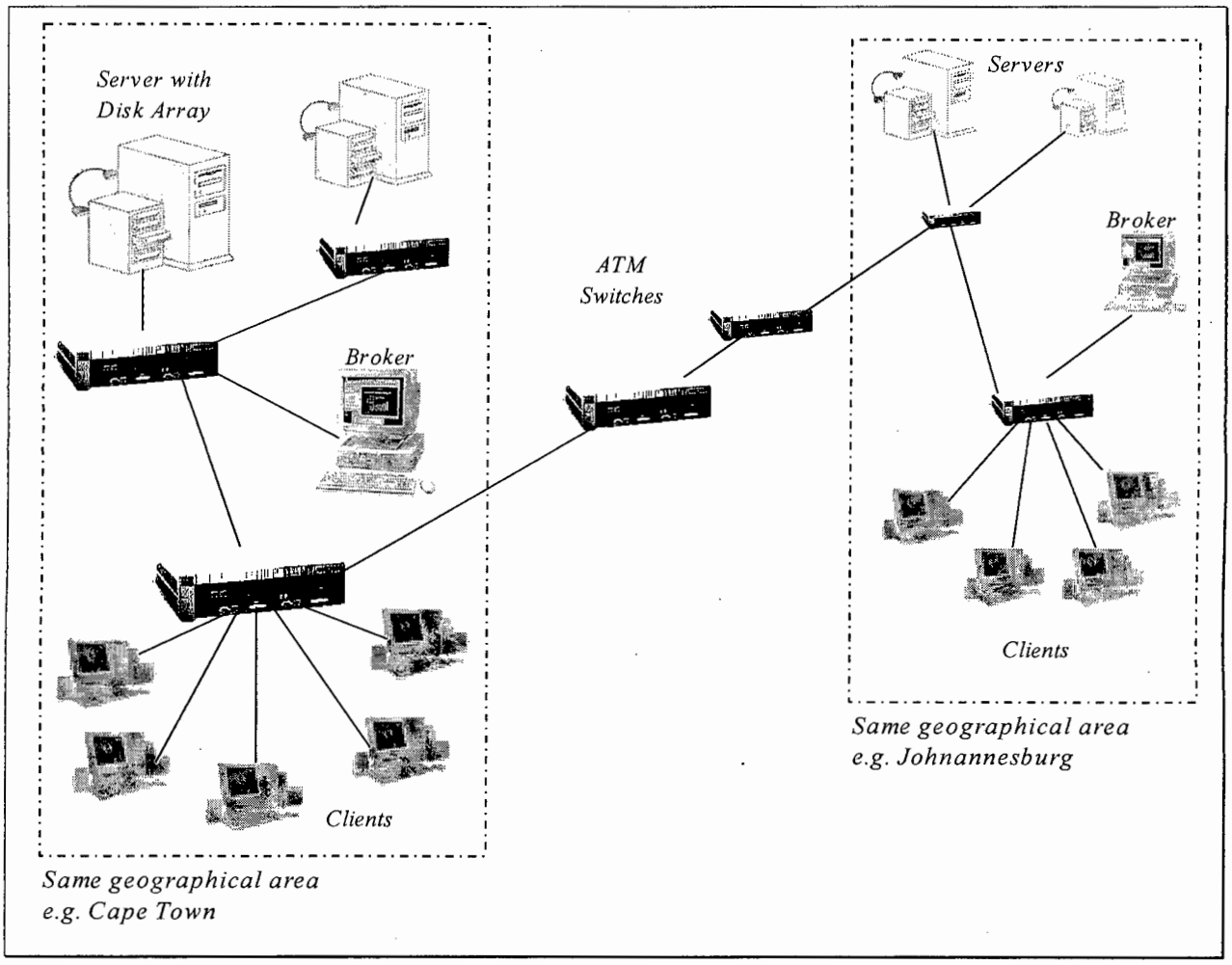

Figure 3-6 Possible physical location of ATOM nodes in an ATM network

3.3.1 Description of the nodes in the ATOM Model

3.3.1.1 Servers

The use of multiple Servers is imperative when providing service to thousands of VoD Clients in a backbone network. More importantly, those Servers must be placed close to the Clients in order to balance network load [16]. ATOM incorporates multiple Servers separated by geographical distance (e.g. a major city could have a video Server per suburb) or content (e.g. some Servers may be devoted to one category e.g. an entertainment video server or an educational video server). This balances the network load since Clients can find a Server close to them, reducing their number of hops through the network backbone to the Server, and can access a Server which they know only entertainment or only educational users are accessing, with similar or predictable access patterns. As an example of this, one might know that high school users will be accessing the Education Server in the afternoon whilst students in Adult Home Education programmes will be accessing the Education Server in the evening, and can thus select usage time accordingly. Each Server contains a disk array as described in Section 4.5.2 of Chapter 4.

3.3.1.2 Brokers

A Broker is a "*dedicated control mechanism that mediates interactions between Client applications requiring services and Server applications capable of providing them.*" [26] A Broker is a software program running on a computer separate from Server machines. A Broker has two main aims. The first is to free Clients from having to maintain information on where and how to obtain particular services. Servers will register their locations with the Broker. The second aim is to share the computational load of processor-intensive operations. In the Client/Server paradigm this implies removing the connection setup/Quality of Service negotiation process and the video search process from the Server to a Broker computer. The Server should be dedicated to streaming video to Clients. Any time taken to service connection requests steals CPU cycles from the video scheduling processes. The aim is to have Broker software which stores information about the state of Server machines in terms of videos available, number of current users, network and CPU resources - therefore each Server must provide regular state updates to the Broker. A Client wishing to participate in a video-on-demand session must first connect to a Broker and state its QoS requirements. The Broker will negotiate a connection to a Server in terms of QoS and video choice. Once the connection is accepted by the Broker, the Client is informed of the ATM address of the correct Server and it will then connect to the Server and start receiving digital video. Since these tasks have been removed from the Server, they do not impact on the resources of the Server machines (allowing the Servers to concentrate efforts on scheduling and streaming videos only). There are two design models for Brokers: forwarding Brokers and handle-driven Brokers [26]. A forwarding Broker relays a Client's request to the relevant Server application, retrieves the response and relay's this to the Client. A handle-driven Broker relays a service "handle" to the Client. This handle contains the name and network address of the Server. The Client can then connect directly to the Server. ATOM uses the handle-driven Broker design. The reason for this is that a direct connection between the Client and Server is required for streaming the video and upstream control signals like play, stop, rewind and fast-forward. One added feature of ATOM is that the Client-Broker and Server-Broker connections are not broken. This enables the Broker to keep track of faults and improve recoverability. Unlike the Anido and Barnett model [17], the ATOM Broker has no cache storage of videos, since then the Broker would be a video Server with management capabilities, negating the reason for separating data and management entities.

3.3.1.3 Clients

Clients interact with Brokers to receive a Server address corresponding to a desired video.

3.3.2 The Video Database

All Servers and Brokers have a video database. Each Server will have a particular set of videos. The Broker's video database will be a merge of all connected Servers' video databases. The video database is implemented with a linked list. Each video item is stored in the linked list and operations such as compare and retrieve are implemented using linked list operations.

3.3.3 ATOM Rules and Protocol

The aim of the distributed system topology is to maximize the Client's chance of finding the desired video. In order to avoid redundancy in this connectivity certain rules are laid down:

1. A Broker must register with all other Brokers so that a mesh of Brokers is formed. These Brokers are then called Peer Brokers. Peer Brokers must share information contained in their video databases. The connections between Brokers are ATM Permanent Virtual Circuits (PVCs). The mesh is fault-tolerant since if a Broker fails, all remaining Brokers are still connected to each other. Although the mesh topology increases the number of links required (as opposed to a bus connection for the Brokers), the increase is tolerable since the total number of Brokers is always very much smaller than the number of Clients and Servers. The gain in distributed information greatly outweighs the extra connections. The mesh topology's most important feature is that it scales well: the number of links increases geometrically with the number of nodes in the system. The mesh can be implemented using ATM multicasting techniques in which use of network resources is minimised by sharing virtual channels where possible.

2. A Server or Client can only register with one Broker at any one time. Any connection to another Broker would be redundant since Brokers are registered with each other.

3. Regular updates of video databases and resource information are required between Server/Broker and Broker/Broker.

The ATOM Protocol is as follows:

1. Brokers register with each other, merging video databases.

2. Servers register with a Broker, providing their video databases. Servers and Clients are encouraged to register with a Broker the minimum number of ATM switches away.

3. Clients request videos from Brokers. Brokers calculate QoS requirements, do Connection Admission Control and inform Clients the location (ATM address) of the Server to connect to. In order to calculate QoS requirements, the Broker consults the video database which will include the average and peak bit rates for each video. This is expanded in Section 3.4.3.

4. a Broker uses the information obtained in the regular Server update (i.e. number of current users, network and CPU resources)

5. Two connections exist between Client and Server. The first connection is for control information (e.g. informing the Server of the movie choice and interactive operations like play, stop, fast-forward and rewind) and the second connection is for downstream video transmission from Server to Client. This ensures that ATOM separates control and transport information as suggested by the specifications in Section 1.4.1.

ATOM can be considered a mixture between a bus and a star topology. The connected Brokers form the bus and the Server-Broker/Client-Broker connections form the stars.

3.3.4 Server Selection Strategy

It is possible for the same video to exist on more than one Server. If this is the case, the Broker must select the most suitable Server based on two criteria:

1. The Server must be closest to the Client in terms of the minimum number of ATM switches between the Client and the Server;

2. The Server with the least load. Each Server should reserve a certain amount of resources to local requests (i.e. from Clients a minimum number of ATM switches away) as in the Chan and Wong model [16].

3.3.5 Protocol if a Broker fails

The question arises of what happens if a Broker machine fails for any reason (e.g. hardware failure). Consider Figure 3-7. Assume the Broker labeled B1 fails, then:

1. Existing Server/Client video stream connections must be maintained.

2. Its Servers and Clients must connect to neighbouring Brokers according to one of the following strategies:

   (i) All migrate to the nearest Broker in terms of ATM Hops. If 2 or more Brokers exist which are an equal number of hops away then one of these Brokers is selected at random.

   (ii) The load is shared amongst the closest Brokers in terms of ATM Hops. If 2 Brokers are closest then the load is divided by 2. If 3 Brokers are closest then the load is divided by 3 etc.

The choice of which strategy is used can be decided by system administrators.

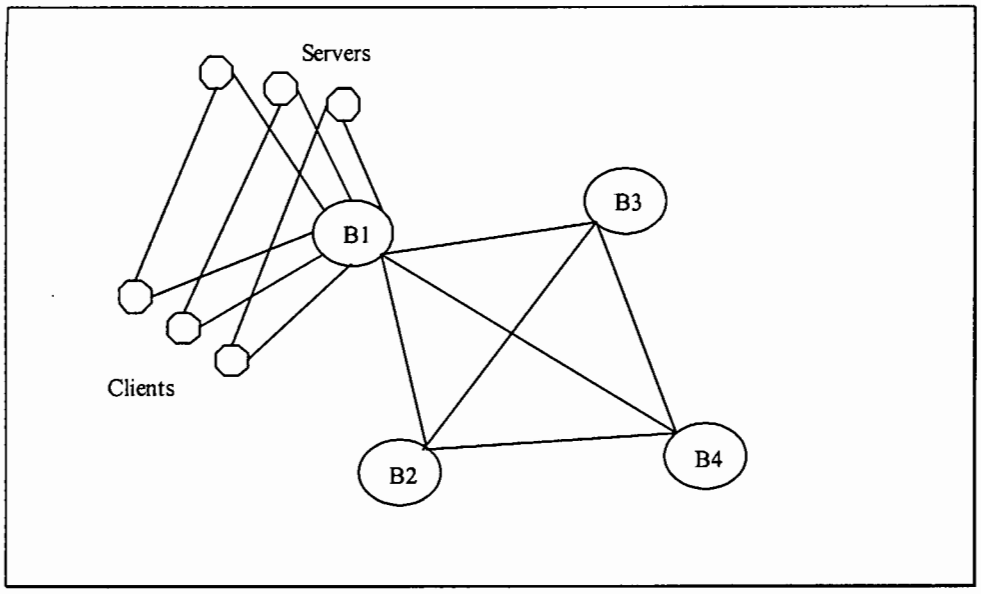

Figure 3-7 Sample ATOM system to demonstrate protocol if Broker Fails

3.3.6 Sample network and example of video selection

Figure 3-8 depicts a sample network built on the proposed topology. With reference to Figure 3-8, the following example is described: Suppose a user on the machine KINGKONG is interested in viewing the video **chicken.mpg**. A search on the Broker it is connected to (Frankenstein) will reveal that **chicken.mpg** is indeed available but on another Broker. The address of the correct Broker will be given to KINGKONG by Frankenstein. KINGKONG will disconnect from Frankenstein and connect to Igor. Igor will then perform

Connection Admission Control and KINGKONG will be passed the ATM address of the Server called Moose in order to start viewing the video from the Moose video Server.

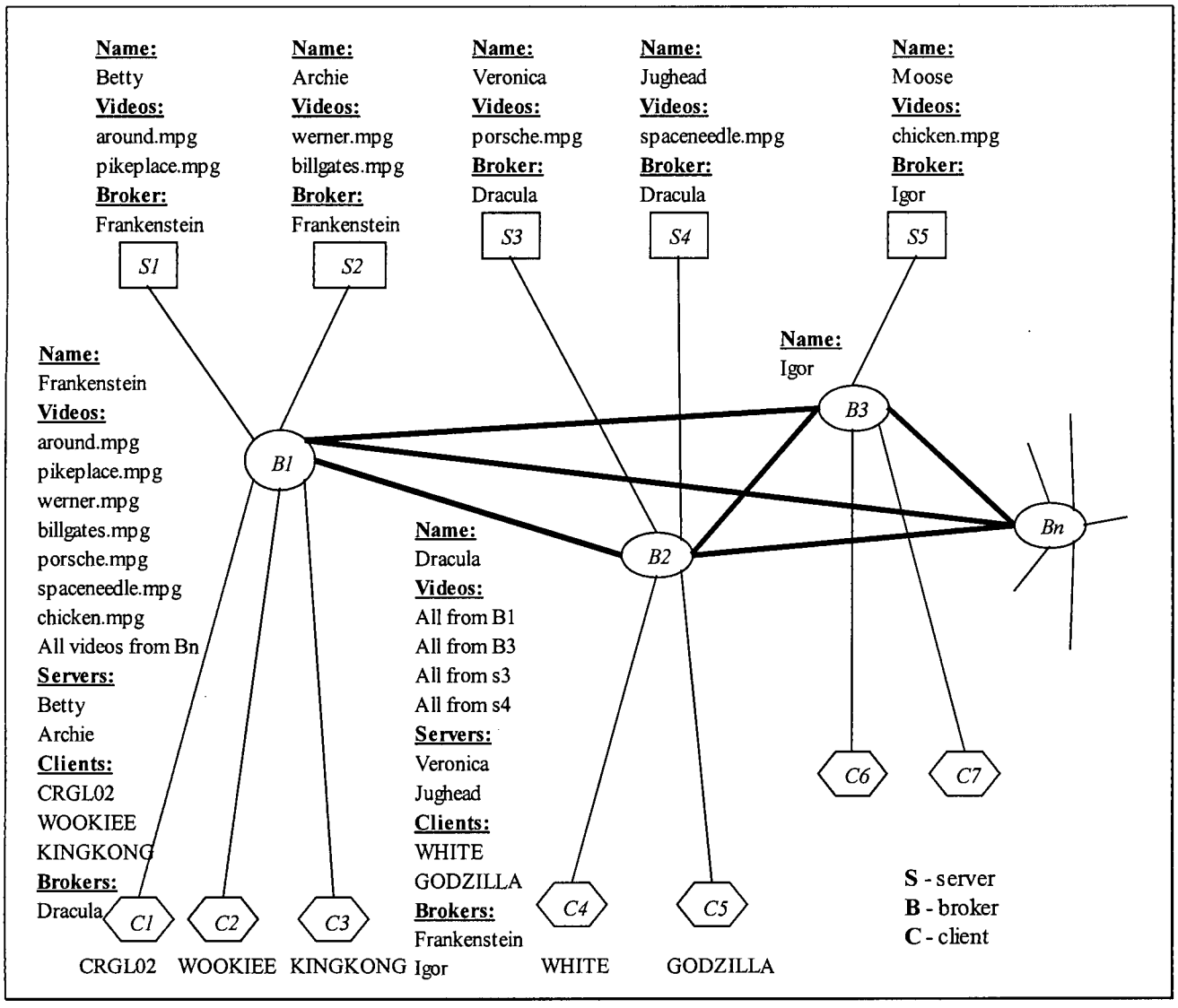

Figure 3-8 Sample network configuration built using the ATOM Model

3.4 Design Issues

A number of critical design issues arise when distributed systems are considered:

3.4.1 How do nodes discover each other?

When a pilot ATM video-on-demand network based on ATOM is launched, there will be a certain number of Brokers, each one serving a number of Servers and Clients. The Brokers will have an NT name and an ATM address. One particular Broker will provide a Web Server with a well-known address (e.g. www.atom-vod.co.za) from which Clients can receive the ATM address of the nearest Broker to them in terms of number of ATM hops. Since the Web Server is hosted on a Broker, the Web Server will have an up-to-date list of all connected Brokers due to the Peer Broker protocol. The Web Server could also supply software upgrades for Clients. The Client would cache information received from a visit to the Web Server, including Broker addresses. This would reduce the number of visits required to the Web Server.

3.4.2 Justification for mesh connection of Brokers and calculation of number of nodes per system

As already stated, the mesh connection of Brokers increases the total number of PVCs required (as opposed to a bus connection for the Brokers). But since the total number of Brokers is always very much smaller than the number of Clients and Servers, the increase is acceptable. The benefits provided by the distributed system greatly outweigh the extra connections. The mesh topology scales well: the number of links increases geometrically with the number of nodes in the system as shown in Table 3 for 10 Brokers. Also, if a Broker fails, its Servers and Clients can be passed to a neighbouring Broker in the mesh as described in Section 3.3.5.

The growth is governed by the formula $\sum_{1}^{n}(n-1)$ where n is the number of Broker Nodes.

Number of Broker Nodes	Number of PVC Connections
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45

Table 3 Number of Broker Nodes versus number of Permanent Virtual Circuits (PVCs)

It is possible to predict the number of Brokers required and therefore the number of PVC connections required in a deployment of an ATOM system. Assume a Broker can manage 25 Servers at one time. This assumption is based on the fact that a Broker serves a geographical area, for example a city. 25 Servers could host full length movies, news, weather and other clips. Further, assume each Server can handle 60 simultaneous Clients. This is based on the figure in Section 3.2.4 provided by the Microsoft Netshow Theater Server system [2] which also consists of the Windows NT operating system and an ATM network. Then each Broker can serve 1500 Clients. Table 4 shows the number of Brokers and PVC connections required against the number of Clients in the distributed system. So to provide service for 15000 simultaneous Clients, 10 Brokers, 250 Servers and 45 inter-Broker PVCs are required. The revenue generated from the 15000 Clients is more than a counter-balance to the cost of the 45 PVCs to the video service provider.

Number of Clients	Number of Brokers	Number of Servers	Number of PVCs
1500	1	25	0
3000	2	50	1
4500	3	75	3
6000	4	100	6
7500	5	125	10
9000	6	150	15
10500	7	175	21
12000	8	200	28
13500	9	225	36
15000	10	250	45

Table 4 Number of Clients, Brokers, Servers and PVCs in a sample ATOM system

3.4.3 How is Connection Admission Control achieved

In Section 3.3.1.2, it was stated that the Broker implements Connection Admission Control (CAC) for the Client. The Broker software must store information about the state of each Server machine in terms of the following parameters which are updated regularly:

1. **Video Database** – The Video Database is sent by the Server to the Broker upon registration and includes all videos on the Server and the average and peak bit rates for each video.
2. **Total Concurrent Accesses Per Video** – This describes how many Clients can access a video concurrently and equals the number of disks in the Server disk array.
3. **Possible Accesses For This Video** – This describes how many Clients can still view a particular video. If no Clients are viewing the video, this parameter is equal to the Total Concurrent Accesses Per Video parameter. For each Client viewing the video concurrently, this parameter is decreased by 1.
4. **Total Concurrent Accesses Possible** – This describes how many Clients can be serviced concurrently by the Server and is a function of the number of disks in the disk array and the total streaming bandwidth of the Server.

Consider the following example (see Figure 3-9): Assume the Server hosts three videos: *terminator.mpg*, *topgun.mpg* and *starwars.mpg* and these are striped over a disk array of 5 disks. Assume the Total Concurrent Accesses Possible parameter is 60. The Total Concurrent Accesses Per Video is 5 (the number of disks in the disk array). The Server registers with the Broker which means it sends the Video Database, the Possible Accesses For This Video parameter (for each video), the Total Concurrent Accesses Possible parameter and the Total Concurrent Accesses Per Video parameter. The Client connects to the Broker and selects *starwars.mpg*. The Broker identifies the Server as hosting this video and checks the Total Concurrent Accesses Possible and Possible Accesses For This Video parameters. Since both are greater than 1, the Client is passed the ATM address of the Server and can connect. While the Server is setting up the streaming video connection to the Server it also sends an update to the Broker, which subtracts 1 from the Total Concurrent Accesses Possible and Possible Accesses For This Video (for *starwars.mpg*) parameters leaving them at 59 and 4 respectively.

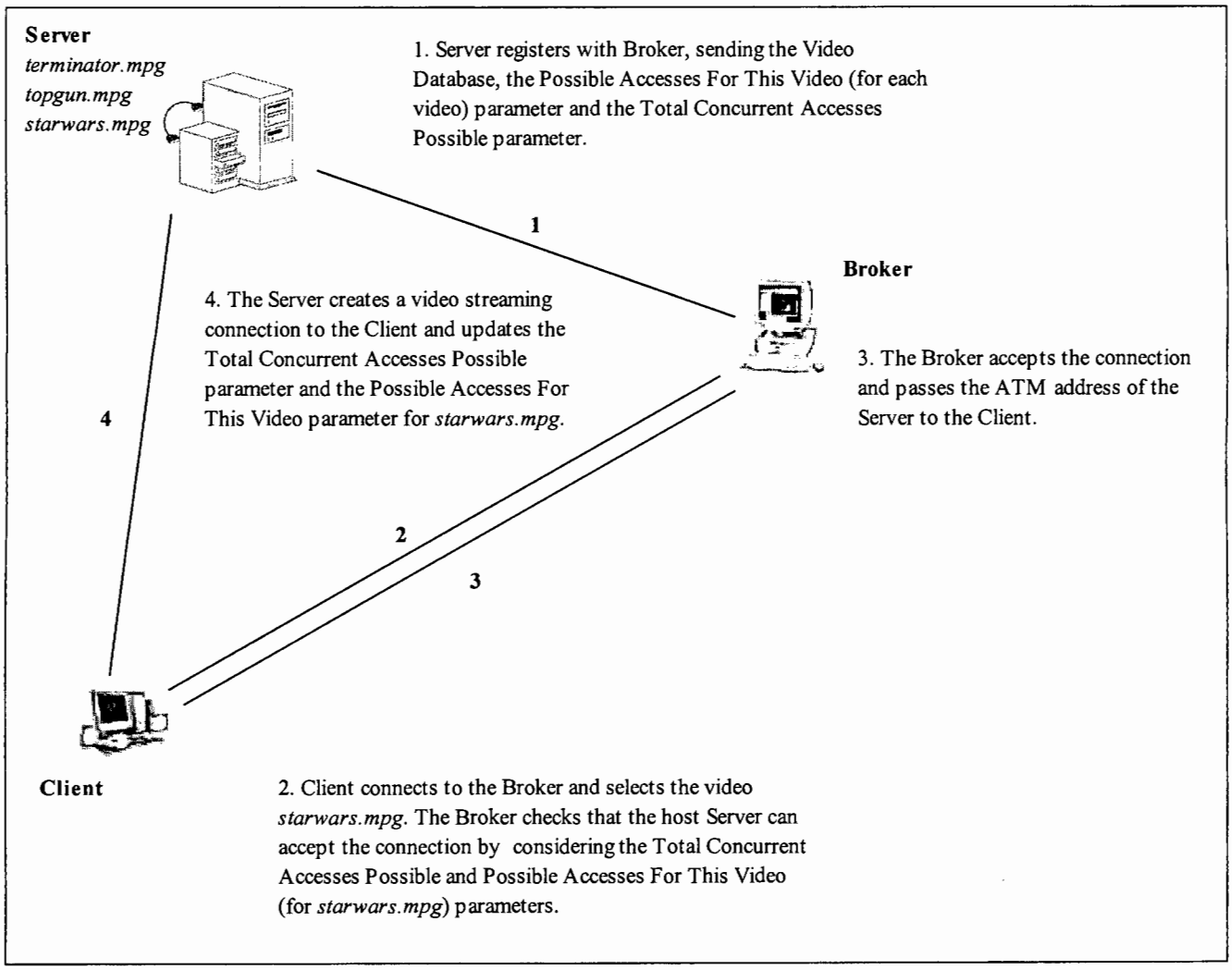

Figure 3-9 ATOM Connection Admission Control Example

3.4.4 How is Customer Billing achieved?

Although customer billing is not within the scope of the design of ATOM, it is worthy of mention since the eventual success of an interactive multimedia system will be measured in financial terms. The basic units of charge in a video-on-demand system are time and bandwidth. We consider the time unit here since bandwidth is on average 1.5 Mbps for each video in ATOM so a fixed charge can be incorporated for this bandwidth. The longer a customer is viewing a video, the higher will be the overall charge for doing so. A certain amount of money per unit of time will be defined, for example, R 0.01 / minute. From an engineering decision point of view, the unit of time selected is of interest. The units available are second, minute and hour. It is not feasible to charge by the second or the hour because the typical length of an entertainment video is about 90 minutes.

Charging by the hour is too great a unit since most videos will not complete more than 2 hours. Charging by the second is overkill since the customer will certainly view a substantial amount of the 90 minute video. Videos should therefore be billed by the minute with different rates depending on video popularity. The second engineering decision concerns where the monitoring and billing take place in the topology. Since the Server is streaming the video over a video connection, it can measure the total number of minutes expended. However, this should then be forwarded to the Broker for calculation and customer billing, leaving the Server free to concentrate on video transmission. A further issue is that customers should receive discounts if they make advance reservation of videos since this enables the Server to reserve the resources in advance. This type of *a priori* information is very useful to connection admission control and resource reservation schemes.

3.4.5 How is Information Security maintained?

There are two ways of considering information security in ATOM. The first concerns customer information and billing and the second concerns the copyright existing on the videos. Customers must have secure login names and passwords and an account must exist on a Broker for the duration of the session so that a Customer can be confident that any bill received will only include videos they have watched. The second area concerns the fact that each video in the system has a copyright which means that the video must not be pirated by the customer. ATOM streams each video to the Client which means that sections of the video are received and played and then discarded in real time. At no stage is the entire video in the customer's memory or hard drive. It is not possible for any video to pirated.

3.5 Advantages of the ATOM Model

1. **Abstraction of Complexity**

   A Client can search for any video on all participating Servers across the entire distributed system. To the Client, it is as if one single Server exists. The underlying network is abstracted away from the Client because video search is always achieved through a single portal – the local Broker. Abstraction of complexity is a fundamental design feature in distributed systems.

2. **Reduces redundant Server connections**

Instead of each Server connecting to each Broker, the Brokers connect to each other. Figure 3-10 shows how 6 redundant connections are eliminated by having one Peer Broker connection. This in turn reduces the network load, decreasing the possibility of congestion in the network.

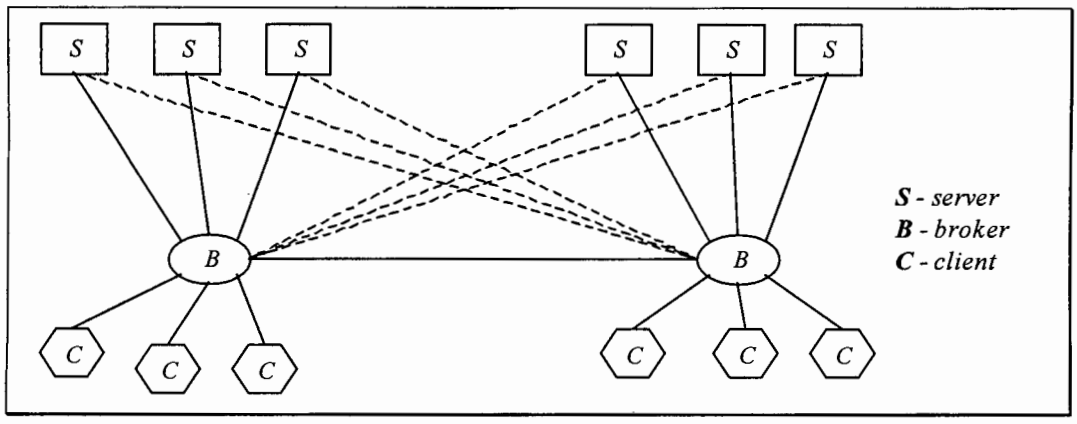

Figure 3-10 Redundant Server/Broker connections - dashed lines are redundant

3. Reduces the number of Switched Virtual Circuits (SVCs)

One of the most important performance parameters in ATM is the time overhead associated with the setting up of SVCs [27] between host and switch and between switches. The FORE ASX200BX switch has an average SVC setup time of 10 ms [27]. However, as the number of connections (or hops) increases, the setup time increases rapidly with the setup time for each hop taking between 45.5 ms and 100 ms [27]. A video-on-demand system consists of SVCs (not Permanent Virtual Circuits) since Client/Server connections are relatively short lived (in the order of minutes and hours). The ATOM Model minimizes the number of connections and therefore minimizes the overhead associated with SVC setup. It does this by encouraging Clients and Servers to connect to Brokers the minimum number of ATM hops away.

4. Improves user Quality of Service (QoS)

The ATOM topology encourages Clients and Servers to connect to the Broker physically closest to them first, thus minimizing the number of ATM network switches traversed and improves the overall QoS available to the user since the video path is shorter. The shorter the path, the less chance of encountering delay, bit errors, buffer overflows and hardware failures in network equipment.

5. **No greater cost**

Anido and Barnett [17] examine the cost of storage in Video-on-Demand Servers. They show that a distributed approach to Video-on-Demand incurs no greater storage cost than a centralized approach.

6. **Separates Control and Transport**

By separating control and transport information, ATOM agrees with the industry specifications in Section 1.4.1.

7. **Video is streamed in real-time to the Client**

Video is streamed to the Client, not downloaded in full and then played. This reduces the storage and memory requirements of a Client and minimizes the delay before playback. Also, the Client is unable to store a video on the local hard drive, thus eliminating any piracy problems. In the all-digital environment characterizing VoD services, information can easily be copied and reproduced. This jeopardizes the established markets of information providers, who lose control of their data.

8. **ATOM uses native ATM**

ATOM uses native ATM, unlike Model 4 which uses Classical IP (CLIP). This eliminates the need for IP packets and CLIP software drivers.

9. **ATOM is scalable**

Adding more Servers increases the total system capacity in terms of storage and bandwidth. Anyone can create a video Server with their own content. They just connect to a Broker and are instantly globally known.

Chapter 4 The ATOM Quality of Service Architecture (AQOSA)

4.1 Introduction

Quality of Service (QoS) defines the effectiveness of the service provided to the user in terms of the multimedia characteristics described in Chapter 2: **bandwidth**, **delay** and **error rate**. If the user of a video-on-demand service is receiving poor QoS, it means that the video and audio are distorted, jittery or unsynchronized. The aim of this Chapter is to first introduce the design methodology for a QoS implementation and then to present the design of the ATOM Quality of Service Architecture (AQOSA). The Chapter begins with a review of current QoS architectures in the literature in Section 4.2. This review highlights important definitions including a flow, service contract and flow management. The principles of Integration, Separation and Transparency are introduced. The issue of mapping human requirements to network QoS parameters is investigated in Section 4.3. The action of a QoS framework is introduced in Section 4.4, including possible causes of QoS degradation. AQOSA consists of 11 modules which work in unison to provide end-to-end QoS guarantees for the video-on-demand streams. These modules are presented in Section 4.5.

4.2 QoS Architecture Review

Most research into providing QoS guarantees in multimedia communications has focused on network-oriented traffic models and service scheduling disciplines. These guarantees are not end-to-end, they only preserve guarantees between the end-system and the network access point. Researchers have recently proposed new communication architectures which are broader in scope and cover all parts of the video stream including the application, the operating system, the communication stack and the network. A number of different approaches are now reviewed and important QoS terminology introduced.

4.2.1 QoS-A

In [15] a system-wide Quality of Service architecture is proposed. The Quality of Service Architecture (QoS-A) is a layered architecture of services and mechanisms for Quality of Service management and control of continuous media flows in multi-service networks. The following key notions are incorporated: *flows* which characterize the production, transmission and consumption of single media streams with associated QoS; *service contracts* which are binding agreements of QoS levels between users and providers; and *flow management* which provides for monitoring and maintenance of the contracted QoS levels. Aurrecoechea et al [28] build on the work in [15] and present the end-to-end QoS scenario shown in Figure 4-1.

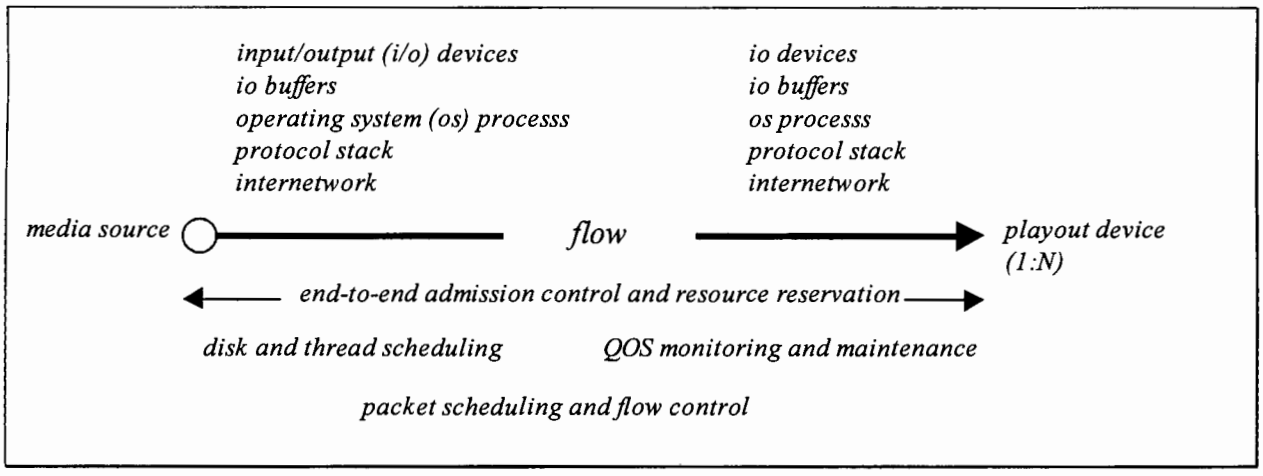

Figure 4-1 End-to-End QoS Scenario (source: [28])

The following points are to be noted from Figure 4-1:

- A flow is depicted. A flow is a single media source (e.g. audio, video or data). A flow traverses resource modules (e.g. CPU, memory, network element) at each layer from source media devices, down through the source protocol stack, through the network, up through the receiver protocol stack to the playout device. The concept of a flow is important because it enables the identification of the areas requiring consideration when designing a QoS architecture. Each resource module traversed must provide QoS configurability (based on a QoS specification), resource guarantees and maintenance of on-going flows. In ATOM, the control and video connections are flows. Note that MPEG-1 video incorporates video and audio into one flow.

- End-to-end Connection Admission Control and Resource Reservation are of prime importance since these can prevent future congestion if applied properly. Connection Admission Control (CAC) accepts or rejects new connections to a multimedia service based on current network and resource conditions. Guaranteeing

QoS relies on reserving resources. In a video-on-demand system there are many resources involved, e.g. the CPU, main memory, bandwidth and network buffers. These resources need to be managed effectively in order to maximize the system. Advance reservation will give the system prior knowledge about projected loads, thus making resource allocation easier.

- The Server machine needs intelligent disk and thread scheduling. Scheduling means to provide each task with enough time in which to be performed. An inefficient scheduling algorithm could result in congestion on the Server even though the network is able to handle more streams.

- The network requires flow control. Flow Control means that a feedback mechanism is in place to inform the source of changes in the network.

- Network and QoS conditions must be constantly monitored to ensure that the delivered Quality of Service is according to the Service Contract drawn up with the Client. This monitoring can be achieved by Usage Parameter Control (UPC) in which the system monitors a predefined set of parameters.

A number of principles motivate the design of a generalized QoS framework as summarised in [28]:

- the *Integration principle* states that QoS must be configurable, predictable and maintainable over all architectural layers to meet end-to-end QoS [29].

- the *Separation principle* states that media transport and control/management are functionally distinct architectural activities [30]. As an example, media transport like video flows require high bandwidth whereas control tasks like signaling require low bandwidth. In ATOM this principle has been obeyed by separating the control and video flows.

- the *Transparency principle* states that applications should be shielded from the complexity of underlying QoS specification and QoS management. In ATOM the Client is not aware of the underlying QoS negotiation.

4.2.2 Heidelberg QoS Model

The HeiProject at the IBM's European Networking Centre in Heidelberg has developed a QoS model which provides guarantees in the end-systems and network [31]. The communications architecture consists of a transport system and an internetworking layer. The transport system provides QoS mapping and media scaling whilst the internetworking layer supports both guaranteed and statistical levels of service. Media scaling matches the source with the receiver's QoS capability. The most important part of the architecture is the HeiRAT (resource administration technique) which includes QoS negotiation, QoS calculation, Connection Admission Control, QoS enforcement and resource scheduling.

4.2.3 TINA QoS Framework

TINA addresses the computational and engineering viewpoints of distributed telecommunications applications. The TINA QoS Framework [32] includes QoS degradation reports if the QoS contract is violated.

4.2.4 Other Architectures

The following related architectures are summarized in Tatipamula et al [7] and are included as further reference for the reader:

1. **XRM** - Extended Integrated Reference Model developed by the COMET group at Columbia University.
2. **OMEGA** - University of Pennsylvania.
3. **Int-Serv Architecture** - Defined by Integrated Services Group of the Internet Engineering Task Force (IETF).
4. **Tenet Architecture** - The Tenet Group at the University of California at Berkeley has developed a family of protocols for an experimental wide-area ATM network.

4.3 Mapping human requirements to Quality of Service Parameters

Quality of Service is implemented using predefined network parameters. These parameters relate to the issues of bandwidth, delay and error rate. Some common QoS parameters are the Peak Cell Rate (PCR) which relates to bandwidth, the Cell Loss Ratio (CLR) which relates to error rate and the Cell Transfer Delay (CTD) which relates to delay. It is important to translate the QoS requirements of the application to the QoS network parameters supported by ATM. The QoS requirements of a video-on-demand application are essentially visual and auditory and are hard to measure objectively since users may have a different concept of acceptable video

and audio playback. The requirements depend mainly on the user's sensitivity to glitches caused by lost cells. Once the requirements have been decided on it is necessary to map these high-level ideas to the network-level parameters like PCR, CLR and CTD. Srivastava et al [22] investigate human factors in interactive viewing in order to understand how these factors can be mapped to network parameters. On a video machine, the time taken by a user to perform an action (e.g. pressing the fast-forward button) is normally much longer than the time taken by the machine to respond but the overall response time is normally less than one second. But a video machine generally doesn't restart at exactly the correct position after a fast-forward or rewind scan. It may start a bit before or after the selected portion. It is possible to extrapolate this to the ATOM system and assume that the user wants almost instant response to the interactive commands but would accept less than perfect restart position in the video. For example, assume the user is playing the video and decides to rewind to review a particular scene. Pressing the rewind button starts the rewind action immediately. Once the user detects the desired scene the play button is pressed and the video begins playback. The user will accept if this playback is slightly before or after the beginning of the scene, depending on the disk access algorithm on the video Server. Srivastava et al [22] contend that the user cannot tolerate jitter in the display, a long wait time in response to a command and losing picture when display is expected. ATOM aims to respond to a user request with no perceivable delay. When play resumes after an interactive command, a position tolerable to the user is found for playback. This is discussed further in Section 5.2.2.1 concerning the implementation of the interactive controls. Another way of determining user requirements is via a tuning service. The user could be presented with a sample video and an interface for input of QoS parameters. Changing the parameters (possibly with a slider bar) will change the quality of the sample video. The user could select a suitable visual and auditory quality and the application would translate this human perception into ATM network parameters.

4.4 The action of a QoS framework

There are two aspects to a traditional QoS framework: applications specify their QoS requirements and network systems provide certain QoS guarantees. This is expanded into the following framework: An application specifies its QoS requirements (e.g. required bandwidth, cell error rate and delay jitter) to the network system in the form of a Service Contract. This Service Contract contains the desired QoS parameters for the connection. The network determines whether it has sufficient resources to satisfy the application's requirements. If it does, the connection is accepted and the local resources are reserved immediately and then committed later if the end-to-end connection admission control test is successful. The end-to-end connection admission control test means that each network device along the path between Client and Server is tested to see if it can satisfy the QoS

requirements of the connection. If the network cannot satisfy the requirements there are two options: reject the connection completely or offer it a reduced resource allocation. If the application is not prepared to accept a reduced resource allocation it can retry at a later stage when more resources are available. A further requirement is an end-to-end resource reservation protocol that can reserve resources on each local resource module traversed (e.g. CPU, memory, I/O devices and switches). The protocol interacts with QoS-based routing to establish a path through the network. Once the Client and Server have agreed on a set of QoS parameters, that set is considered to be the Conformance Definition by which the Server can police the connection using Usage Parameter Control (UPC), to ensure no abuse of the Service Contracts occurs. During the lifetime of the connection, any parameter that does not agree with a value in the Conformance Definition is abusing the Service Contract. A further issue to decide upon is whether the QoS framework should allow for in-service QoS renegotiation or not. This implies that the system will allow the Client and Server to change the values of the QoS parameters in the Service Contract dynamically in the event of network congestion or degradation. If this is not supported, then the Client must undergo a disconnection/re-establishment phase. It is worth implementing QoS renegotiation if the system has a high Client turnover rate i.e. many Clients logon or logoff frequently, with the implication that the system resources are in demand. The video-on-demand system does have the potential to be a highly dynamic application environment and therefore must be designed for in-service QoS renegotiation. The implication is that the system must constantly monitor resources in order to detect system degradation. With this warning it can renegotiate the Service Contracts with Clients. Campbell et al [15] include QoS renegotiation in the QoS-A design. CBR and VBR services do not support bandwidth renegotiation after a connection has been set up but ABR does offer the possibility of bandwidth negotiation using resource management (RM) cells. Atiquzzaman and Zheng [33] consider transmitting compressed multimedia over the ABR feedback-based service of ATM. They show that the feedback technique can be used successfully in traffic management of multimedia streams.

Having presented the traditional use of the Service Contract in the QoS framework, it must be noted that the ATOM design uses the Service Contract in a different manner. In a video-on-demand system, the Client is unaware of the network requirements of a particular video. This information is however known to the Server. A Client is unable to abuse the Service Contract because it can only play the video at predefined normal and fast rates of the system. However, the Server is capable of abusing the Contract by providing the Client stream with too little bandwidth or too much error or delay. The Service Contract should therefore be defined at the Server and be used by the Server as a way of policing the Quality of Service it is providing to the Client.

4.4.1 Possible causes of QoS degradation

Having introduced the concept of QoS, it is necessary to note what factors could lead to impairment of the QoS available for the connection. The following areas are possible factors:

- Propagation delay of cells through the network caused by the physical media. For a metropolitan area network this is equal to a few milliseconds. For a wide area network this is equal to a few hundred milliseconds. This delay is constant for each cell in a particular connection since the distance is fixed for that connection;
- Random and/or bursty bit errors caused by the physical media;
- Switch Architecture, specifically a switch with insufficient buffer capacity;
- Traffic Load caused by other video streams;
- Number of Nodes in Tandem. This refers to the number of ATM switching nodes the connection traverses. The larger the number of nodes, the greater the chance of QoS degradation;
- Poor resource allocation strategy;
- Hardware failures including port failures, switch failures or link failures;
- Cell Loss and Cell Misinsertion. One possible cause of Cell Loss is congestion caused by statistical multiplexing.

4.5 AQOSA

The design of the ATOM Quality of Service Architecture (AQOSA) is now presented in Figure 4-2 and detailed in Sections 4.5.1 through 4.5.12. AQOSA consists of 3 sections: the Server side, the Internetwork and the Client side. The aim of AQOSA is to enable end-to-end Quality of Service guarantees for all streams in the ATOM system. A video connection between the Client and Server is a flow as defined in Section 4.2.1. Each part of the architecture has a role to play in maintaining the guarantee.

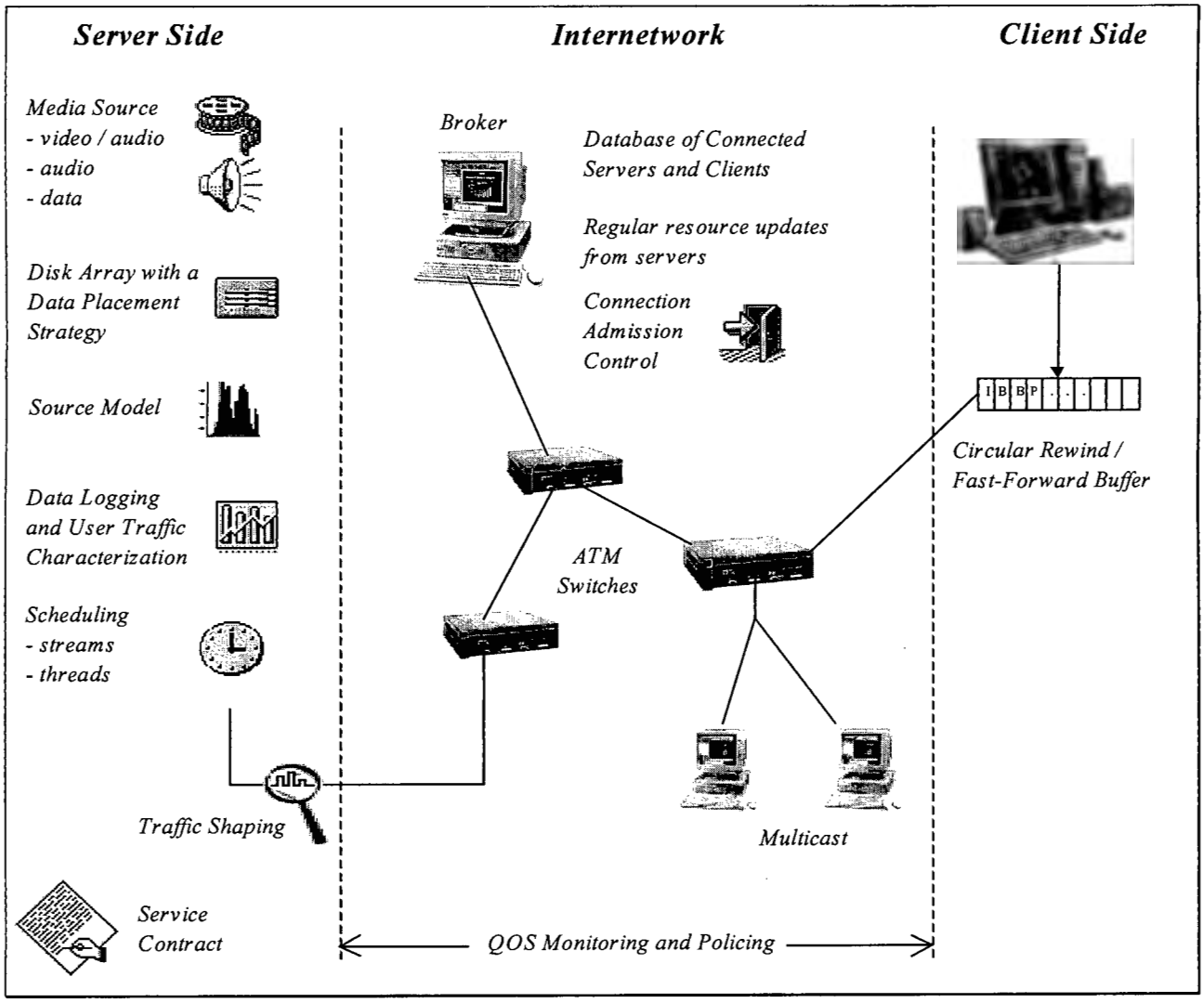

Figure 4-2 The ATOM Quality of Service Architecture (AQOSA)

4.5.1 Media Source

This refers to the data requested by the Client and could be in the form of video with audio, only video, only audio or large graphic files. It is important that this data has peak bandwidth significantly smaller than the total streaming bandwidth of the Server. If each stored video has bandwidth similar in size to the total bandwidth of the Server, it is clear that very few concurrent streams will be possible. This will also increase the likelihood of congestion. For example, assume that the Server has a total streaming bandwidth of 50 Mbps. If a video stored on this Server has a playback bandwidth of 10 Mbps then this would be far too high for the system since only 5 streams of this video could be played concurrently. If the video was restricted to a bandwidth of 1.5 Mbps then

33 streams of the video could be played concurrently. It can therefore be seen that intelligent choice of videos in respect of peak bandwidth can improve overall system capacity.

Archived applications like video-on-demand have an obvious advantage over videoconferencing in that the video is encoded and compressed offline and stored on the mass storage medium for later transport. Videoconferencing compresses the video on the fly and offers it to the network, making resource reservation difficult since frame sizes and bit rates are not known in advance. With VoD, the system has *a priori* knowledge of the video and can use this traffic profile to develop an appropriate bandwidth allocation scheme.

4.5.2 Data Placement Strategy on physical storage media

In order to be profitable, a distributed video-on-demand system must support thousands of simultaneous viewers, requiring massive throughput from each Server. All the Clients of a particular Server may be watching the same movie, different movies or different parts of the same movie. An effective data placement strategy (DPS) ensures the maximum speed of data transfer from the physical storage devices (e.g. hard disk or CD-ROM) and enables multiple simultaneous streams from the same stored video. The DPS must incorporate multiple disks in a disk array. A well-designed DPS can facilitate the use of interactive VCR commands without an increase in consumed disk capacity. A disk has many latencies including seek time, rotational latency, bus contention and bus transfer rates. These must be minimized in order to ensure data moves to the network card as quickly as possible. A further aim is to avoid disk "hot spots". A "hot spot" arises when one disk in the disk array is saturated with I/O requests while the rest lie idle. This can be removed by balancing the load across the disks. VCR commands can produce very unbalanced disk accesses and subsequent "hot spots". In this section a DPS is designed for ATOM. The DPS consists of the following:

- disk striping of the videos across the disk array, optimised for sequential access;
- a fixed fast-forward/rewind speed and a prime number of disks to completely eliminate hot spots.

4.5.2.1 Disk striping

Disk striping is a technique for spreading data over multiple disks. A disk file system called a stripe set is created by dividing data into blocks (called strips) and spreading them in a fixed order across all disks in the array. For Windows NT Workstation and Server, the size of the strips is 64 KB. It will be shown that this is not a

convenient value for variable bit rate video and that a larger strip size is required. A stripe set is similar to a table, where a disk is a column and strip is one of the entries in the table. A stripe is all of the entries in one row. Figure 4-3 depicts a stripe set, showing the order in which data is written to the set. When data is written to a stripe set, the data is written across the strips in the volume. This ensures the placement is optimised for sequential access as opposed to random access. Chaney et al [34] show that the best way to get optimum performance from disk is to stream large amounts of sequential data (by having a large strip size), to allocate I/O buffers so that transactions are as large as possible and to minimise the amount of disk seeking.

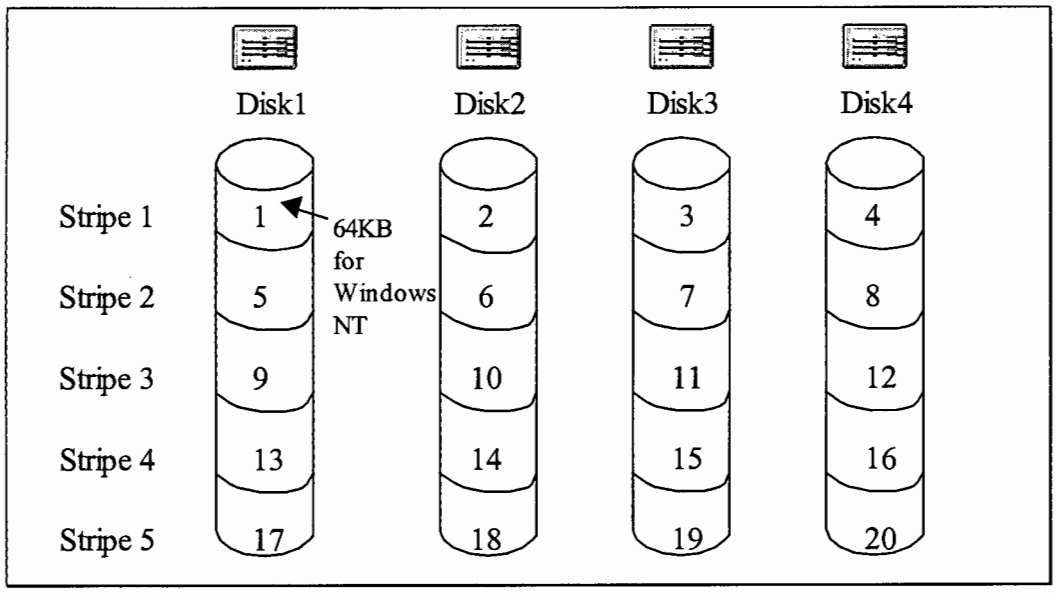

Figure 4-3 Disk striping over 4 disks showing the order that data is written to the set

Disk striping can speed up retrieval operations since the retrieval can be done from each disk in parallel, increasing the aggregate disk bandwidth. The disk array requires a controller. A common controller method is RAID (Redundant Array of Inexpensive Disks) of which the two most applicable levels are Level 0 and Level 5. Level 0 provides disk striping with no redundancy for error recovery. Level 5 spreads data and parity information across all the disks in the set, thus allowing for recovery and continued operation of the disk set if a single drive fails. RAID can be implemented in hardware or software. Hardware implementations are expensive but don't compete for processor cycles. Software implementations use system processing resources but are cheap. A comparison of Level 0 and Level 5 is required in order to choose the most effective level for AQOSA. Level 0 provides the highest disk I/O performance of all disk management strategies but does not provide fault-tolerance protection - if one disk fails, all the data on the stripe set are inaccessible. Level 5 provides redundancy

but the parity information takes up space on each disk equivalent to 1/Number of Disks. When a member of a stripe set with parity fails in a severe manner, such as a loss of power or a head crash, it is possible to regenerate the data for that member of the stripe set from the remaining members. RAID Level 0 is chosen for ATOM because of the disk I/O performance and the fact that no disk space is used to store parity information. In order to provide recoverability, each disk can be mirrored if required.

For a video-on-demand Server, the video file is broken up into segments which are distributed in a round robin fashion across the available disks. This allows multiple users to access the video file simultaneously (although not the same segment at the same time). This is more advantageous than having copies of the same video on multiple disks, as doing that you get the same number of users, but D times the disk space usage, where D is the total number of disks used. There is a practical disadvantage to disk striping, as noted by Vine [35]. Disk striping provides a fixed strip size which suits constant bit rate video because it has fixed size frames. However, the ATOM media is variable bit rate MPEG-1 video (between 1.5 and 4 Mbps), producing frames of varying sizes. A convenient striping segment is one Group of Pictures (GOP) which is 15 frames beginning with an I-frame. It is convenient because the playback after rewind, fast-forward, pause or stop starts with an I-frame. The following calculations show the storage requirements per GOP for the MPEG-1 variable bitrate range:

If the current video rate is 1.5 Mbps at 30 fps then the storage required per GOP is

$$\frac{1.5Mbps}{30fps} * \frac{1}{8} * 15\,frames = 98304 \text{ bytes} = 96 \text{ KB}$$

If the current video rate is 4 Mbps at 30 fps then the storage required per GOP is

$$\frac{4Mbps}{30fps} * \frac{1}{8} * 15\,frames = 262144 \text{ bytes} = 256 \text{ KB}$$

Window NT provides a maximum strip size of 64 KB which is too small to cater for these storage requirements since then each GOP would have to be striped over more than one segment. A file system is required which can cater for 256 KB strips and thus one GOP is always mapped to one strip. This is the same strip size used in the Tiger Shark file system [36]. Therefore, ATOM consists of a disk array with a file system providing strip sizes of 256 KB controlled by a RAID Level 0 Controller.

4.5.2.2 Fixed fast-forward/rewind speed

VCR commands can produce clustered disk access in simple round robin striping since a particular FF-RW playback rate might retrieve all its segments from a single disk in the array. To illustrate this, assume a video file consisting of 45 segments is striped over 9 disks in a simple round robin manner as shown in Table 5. The 9 disks are labeled $D_0 \ldots D_8$, the 45 video segments are labeled 0...45 and the playback speed is marked as a superscript. Table 5 shows that a playback speed of 3x requires accessing disks D_0, D_3 and D_6 repeatedly, producing hot spots on those disks.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
0^3	1	2	3^3	4	5	6^3	7	8
9^3	10	11	12^3	13	14	15^3	16	17
18^3	19	20	21^3	22	23	24^3	25	26
27^3	28	29	30^3	31	32	33^3	34	35
36^3	37	38	39^3	40	41	42^3	43	44

Table 5 Example of hot spots in simple round robin disk striping

However, a playback speed of 4x would present a balanced load to the disk array as shown in Table 6. The playback speed is marked as a superscript.

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
0^4	1	2	3	4^4	5	6	7	8^4
9	10	11	12^4	13	14	15	16^4	17
18	19	20^4	21	22	23	24^4	25	26
27	28^4	29	30	31	32^4	33	34	35
36^4	37	38	39	40^4	41	42	43	44^4

Table 6 Example of balanced load in simple round robin disk striping

These examples show that although simple round robin disk striping is easy to implement, it is not suitable for arbitrary-rate playback since certain playback speeds produce disk hot spots. In the ATOM system, it is not

necessary to provide arbitrary rate playback since ATOM models a video cassette recorder in which the user is provided with one rewind and fast-forward speed and perhaps a slow rewind and slow fast-forward speed. By choosing particular playback speeds and a prime number of disks, it is possible to eliminate disk hot spots completely. The need for a prime number of disks is proposed by Kwon et al [37]. Their Prime Round Robin Disk Striping (PRRDS) method uses a prime number of disks for striping, ensuring that no "hot spots" occur across the disks, even for arbitrary rates of playback. PRRDS offers arbitrary playback rates with the exception of any multiple of the prime number. For example, for a disk array with 5 disks, 1x, 2x, 3x, 4x, 6x, 7x etc. playback can be attained with no change to the load balancing of the disk array and no single disk gets overloaded with requests. Overloading would occur if a request for 5 times playback of the video were requested as the Server would be accessing disk 1 continuously (assuming that disk 1 was the starting point before the request). Since 5 is a multiple of the prime number in this case (multiple = 1) the PRRDS technique excludes its use.

4.5.2.3 Example disk array for an ATOM Server using the Data Placement Strategy

Consider a Server in the ATOM model. It must host 30 MPEG-1 movies of 1½ hour each, requiring a total of $(30*\frac{1.5*1024*1024*90*60}{1024*1024*1024}*\frac{1}{8}) \approx 30$ gigabytes of storage space. Assuming each disk in the disk array can store 8 GB, a convenient prime number of disks is 5, providing a total of 40 GB of storage and the possibility of 1,2,3,4,6,7 etc times playback speeds (not 5x since that violates the prime number constraint). Disk hot spots are completely eliminated. It now remains to choose a playback speed for normal and slow fast-forward and rewind. The normal FF-RW speed is chosen as 4x and the slow FF-RW speed is chosen as 2x normal playback. Figure 4-4 depicts the DPS for this Server. Each strip is labeled ViSi where Vi means video i and Si means segment j (where i = 1,2,...100 and j = 1,2,...). For normal playback of video Vi, the segments are retrieved in the order Disk1,Disk2,Disk3,Disk4,Disk5. For normal fast-forward of Video Vi, the segments are retrieved in the order Disk1,Disk5,Disk4,Disk3, Disk2. For normal rewind (assuming starting at Disk1 for some segment number in the video) the segments are retrieved in the order Disk1,Disk2,Disk3,Disk4,Disk5. For slow fast-forward the segments are retrieved in the order Disk1,Disk3,Disk5,Disk2,Disk4. For slow rewind (assuming starting at Disk1 for some segment number in the video) the segments are retrieved in the order Disk1,Disk4,Disk2,Disk5,Disk3. At no time does an interactive control cause disk hot spots.

How many streams can this Server sustain? Assume each disk has a transfer rate of 2.7 Mbytes/s. Each stream requires 1.5 Mbps which is 192 Kbytes/s. So each disk can handle 14 streams. Therefore with 5 disks in the disk array, this Server can sustain 70 concurrent streams.

The disk array has an aggregate bandwidth of 13.5 Mbytes/s = 108 Mbits/s. The ATM network adapter offers up to 155 Mbps so by sourcing disks with a better transfer rate, it is possible to increase the number of concurrent streams.

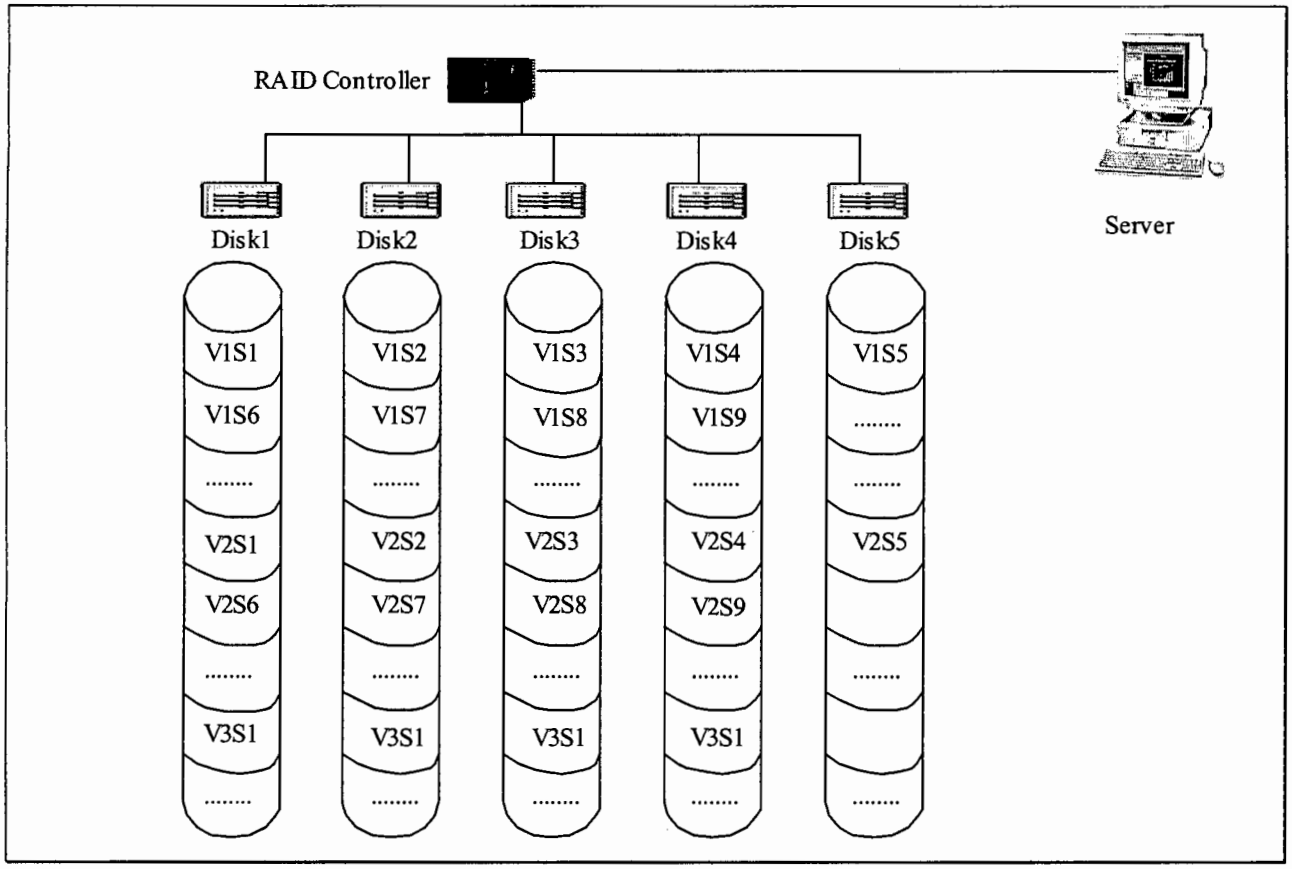

Figure 4-4 Sample Data Placement Strategy for an ATOM Server with "hot spot" elimination

4.5.3 Source Model

A source model that captures the essential features of traffic helps enhance network utilization since it can provide the parameters used in Connection Admission Control (CAC) and Usage Parameter Control (UPC). A video source model will consist of a set of parameters that can be used as traffic descriptors for the CAC and

UPC of video connections. Most of the existing models can be classified as either Markov Models or Autoregressive models. Markov Models partition the peak transmission rate into N equal transmission rates, with each rate representing a state in the Markov chain. In the Autoregressive Model, the bitrate of the next frame is predicted based on the correlation between two or more previous frames. Lele and Nandy [38] describe a Markov Chain source model for VBR video based on entropy.

Rose [16] presents measurements for different categories of MPEG-1 videos. He shows that typical television sequences like sports, news and music videos encode to MPEG-1 sequences with a high peak bit rate and a high peak-to-mean ratio compared to encoded cinema movie sequences. This is a result of the increased data needed to represent the rapid movements of a lot of small objects in the television sequences (e.g. players moving in a soccer game). However, Rose shows that the statistical properties of the sequences of the same category, for example sports, are not necessarily similar enough to make generalizations about a traffic model for that category. This implies that defining traffic models for different categories of MPEG-1 videos is a non-trivial task. Tsang et al [14] also state that video sequences with complex spatial-temporal activity (e.g. high action scenes, scene changes, pans and zooms) require larger bandwidth since less compression is possible. Rose [16] also shows that correlations exist on a frame-by-frame and a GOP-by-GOP level. Due to the nature of the I, B and P-frames available in MPEG there is a short-term dependence on the particular GOP that is used. This short-term auto-correlation is relatively fast-decaying. However there is also a long-term auto-correlation due to the content of the video which is a slow-decaying auto-correlation. The literature [16], [39] suggests that a hierarchical or layered approach is needed to model MPEG-1 video traffic. It has been suggested to classify the video traffic in terms of the different time scales involved. This would lead to modeling the video source in terms of the cell layer, the frame layer, the GOP layer and the call/connection layer. These have durations of the order of microseconds, milliseconds, seconds and minutes respectively. The cell layer model would depend on the ATM Adaptation Layer (AAL) used and the presence of traffic shapers at the ingress to the network. The frame and GOP sizes can be approximated by Gamma or Lognormal probability density functions (PDFs) as described in Rose [16]. The ATOM Source Model defines the source as variable bit rate MPEG-1 video. A comprehensive model for VBR MPEG-1 traffic is provided by Krunz and Tripathi [15].

4.5.3.1 Derivation of the B-PR parameter for MPEG-1 video traffic

It is now proposed that the amount of motion in a video can be used to derive a useful parameter for the Traffic Contract. Movies with a high degree of motion will have larger peak bit rates and larger average bit rates because the B-frames will be bigger. The B-frames only increase in size if the predicted (i.e. P-frame) image is poor due to high motion. Therefore the concept of a B-P Ratio (B-PR) is introduced. The B-PR is the ratio of the total size (in bytes) of the B-frames to the total size (in bytes) of the P-frames for a GOP, normalized by multiplying by the ratio of P to B-frames in the particular GOP of the video. A B-PR file is a very small file in the order of 10 Kbytes containing the B-PR for each GOP in the video. The B-PR file will be included in the Service Contract as discussed in Section 4.5.6. Equation 1 presents the B-PR. In Equation 1, the following key applies:

- ΣB means the sum of the sizes of all the B-frames in the GOP (in bytes or bits)
- ΣP means the sum of the sizes of all the P-frames in the GOP (in bytes or bits)
- P means the number of P-frames per GOP
- B means the number of B-frames per GOP

$$B-PR=\frac{\Sigma B}{\Sigma P}\times\frac{P}{B}$$

Equation 1 The normalized B-P Ratio

As an example of the use of the B-PR, consider the MPEG file **porsche.mpg**. Its GOP structure is IBBPBBPBBPBBPBB. For every GOP, $P = 4$ and $B = 10$. Table 7 presents the frames sizes for the first GOP in **porsche.mpg** and shows the calculation of the B-PR for the first GOP.

Frame Type	Size in Bytes
I	10884
B	3822
B	3806
P	7896
B	4014
B	3486

P	7555
B	4014
B	6087
P	6382
B	4206
B	4275
P	9175
B	3886
B	4807
$B-PR=\frac{42403}{31008}\times\frac{4}{10}=0.54699$	

Table 7 Calculation of the B-PR for the first GOP of porsche.mpg

A B-PR over 1 means that there is more B information than P information which means that there is high motion in that GOP. A BPR below 1 means that there is more P information than B information, and thus less motion is represented in that GOP. This parameter can be used to class videos into three categories: High Motion, Medium Motion and Low Motion. Video content with probable High Motion includes sport and cartoons. General TV series, TV movies and music clips have Medium Motion probability. Cinema movies and TV "talking-head" programs have Low Motion probability.

An MPEG-1 video produces bursts at the beginning of each GOP due to the presence of a large I-frame. A consequence of larger B-frames present in High Motion videos is that the network will receive more video per GOP from the video source, thus increasing the network load because of this increase in the burst size. So the B-PR will help the Connection Admission Control algorithm to decide whether the Client should be allowed to view the video. In fact, the B-PR values for the video can be used by the Server to accurately predict bandwidth changes. Every GOP has a B-PR and since a GOP represents 0.5 seconds of video (for 30 fps video and a GOP of 15), the Server has a useful time in which to make future bandwidth requirement calculations. Simply by looking ahead at the B-PR file for a currently streaming video, the Server can predict the bandwidth required by that stream in the future in time slices of 0.5 seconds. Figure 4-5 presents the B-PR file for **porsche.mpg**. This video has 309 frames (21 GOPs where the GOP pattern is IBBPBBPBBPBBPBB) and is 1966082 bytes large. The following can be deduced from Figure 4-5:

- The video does not have excessive motion (since the B-PR is well below 1) and therefore will not have large B-frames.
- The motion seems periodic, since there are regular peaks (at 2, 5.5, 9, 12.5,16 and 21). On viewing the video, the author feels that this is in agreement with the periodic scene changes. This is a subjective decision but was confirmed by a colleague.
- The Server would not have to be concerned about this video stream presenting long bursts of data.

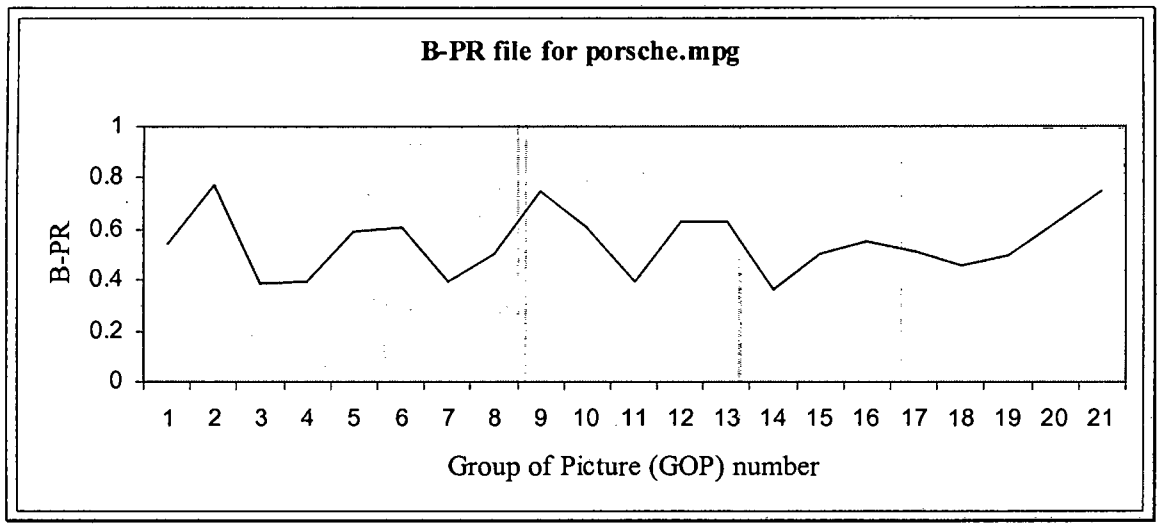

Figure 4-5 B-PR file for porsche.mpg

In order to confirm the usefulness of the B-PR file it is necessary to compare long video sequences that have been encoded using the same parameters. For the performance analysis of communication networks carrying video traffic, Rose [16] has made available the frame size traces[1] (in bits) for a number of videos in the following categories: **movies** (bought on video cassette), **TV sports events** (recorded from TV), **TV sequences** like movies, talkshows, cartoons and music (recorded from TV). The sequences were captured from a VCR (VHS) with a SunVideo SBus board using Motion-JPEG capture at 25 fps. Berkeley MPEG Encoder software (v1.3) running on a Sun Sparc 20 encoded the final MPEG-1 stream from the Motion-JPEG. All the sequences were encoded using the following set of parameters:

[1] These MPEG video traces are available via anonymous FTP at:
ftp://ftp-info3.informatik.uni-wuerzburg.de/pub/MPEG/

- GOP Pattern: IBBPBBPBBPBB (12 frames, $P = 3$, $B = 8$);
- Encoder input: 384 x 288 pels with 12 bit colour information;
- Motion vector search: logarithmic/simple;
- Quantization values: I=10, P=14, B=18;
- Number of frames per sequence: 40000 (about half an hour of video).

The Author programmed a software tool (called ToolBox) to analyze the traces in order to calculate the B-PR for each GOP and thus create a B-PR file for each video. The source code for this ToolBox is available in Appendix A and on the accompanying CD. The first 2000 GOPs were analyzed for each file. This equates to 30000 frames. Table 8 lists three of the videos analyzed, showing their content and also the calculated average B-PR. Figure 4-6 shows the entire B-PR file for **soccer_2**. Figure 4-7 shows the first 144 GOPs for **soccer_2**, while Figures 4-8 and 4-9 show the entire B-PR files for **lambs** and **news_2** respectively. The following can be deduced from Table 8 and Figures 4-6, 4-7, 4-8 and 4-9:

- **soccer_2** has the highest average B-PR, indicating that it has the highest motion.
- There are a number of peaks in **soccer_2** (Figure 4-6) that indicate areas of high motion and therefore high bandwidth. The Server could make use of this in its bandwidth allocation strategy.
- Both **lambs** (Figure 4-8) and **news_2** (Figure 4-9) have a very large peak. The Server would be aware of this peak and would be able to take action.

Video Name	Source	Content	Average B-PR
soccer_2	TV	World Cup 1994: Germany - Belgium	1.13360
lambs	Cassette	Silence of the Lambs	0.955756
news_1	TV	German TV News	0.998733

Table 8 Details for video traces of Figures 4-6, 4-7, 4-8 and 4-9

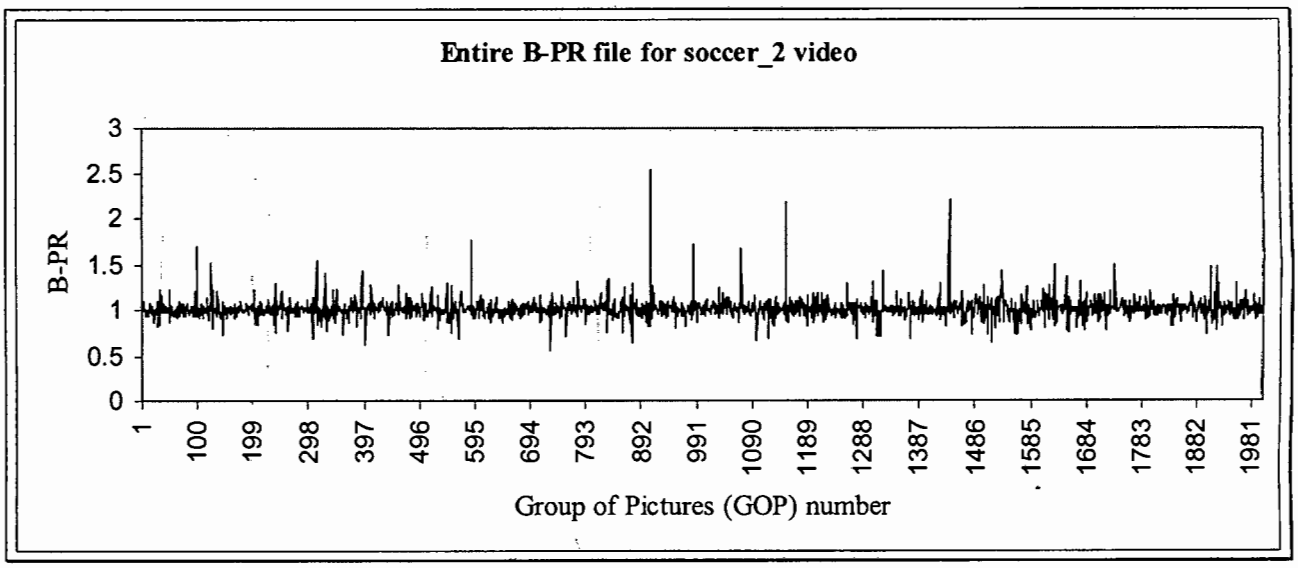

Figure 4-6 Entire B-PR file for soccer_2 video

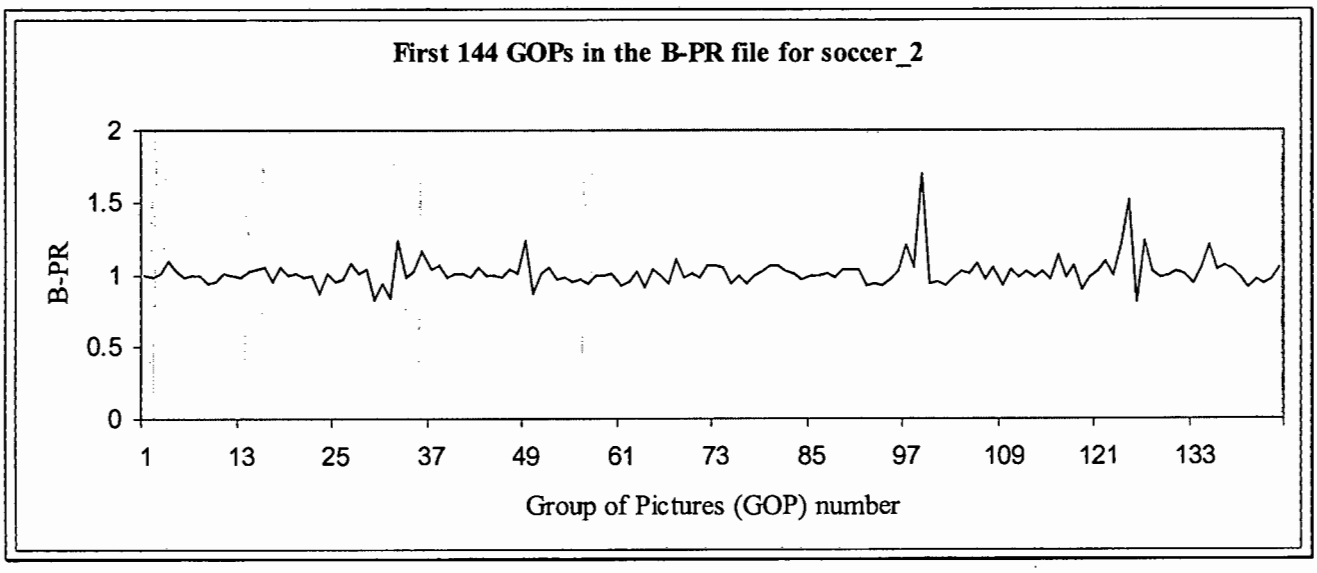

Figure 4-7 First 144 GOPs in the B-PR file for soccer_2

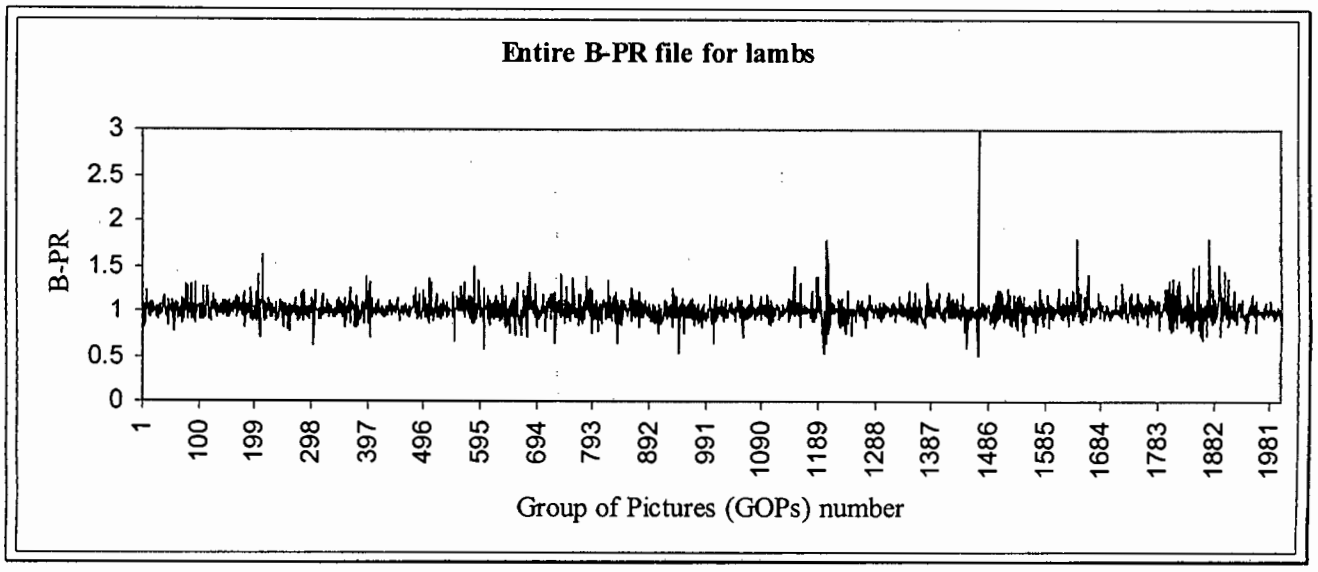

Figure 4-8 Entire B-PR file for lambs

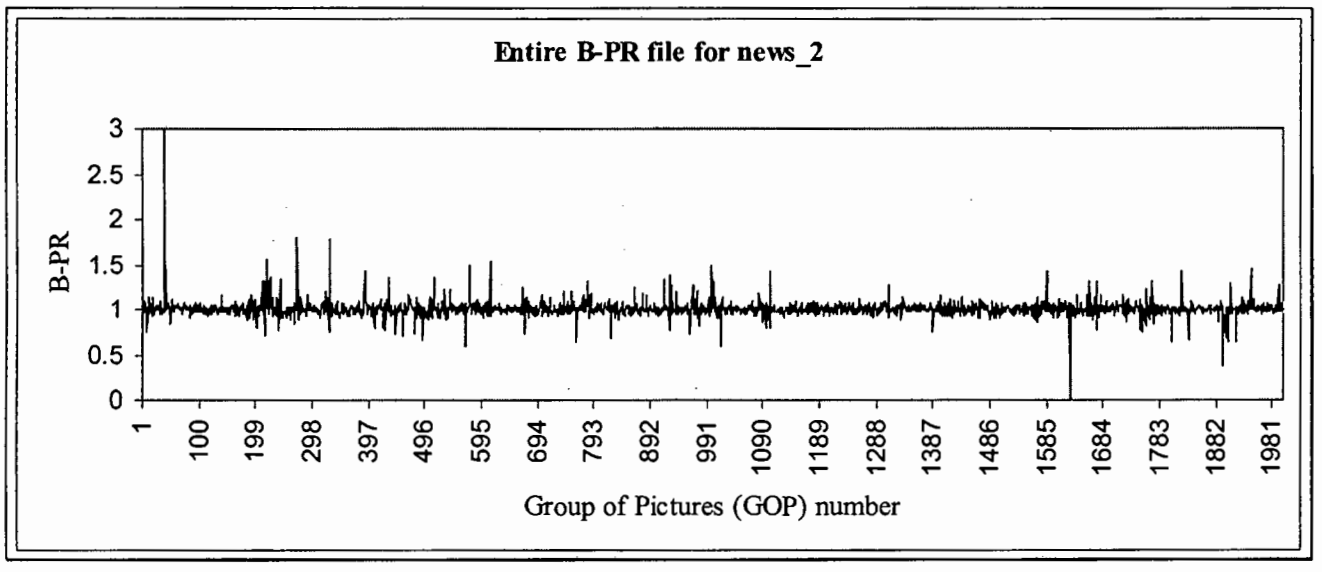

Figure 4-9 Entire B-PR file for news_2

4.5.4 Scheduling of streams and threads

A media stream's continuous playback is a sequence of periodic tasks with deadlines. The tasks are retrieval of media blocks from disk, and the deadlines are the scheduled playback times. The Server's challenge is to supply the stream buffers with enough data to ensure that the playback processes do not starve. The ATOM Server has an algorithm for scheduling packets of video to each Client on time. An inefficient scheduling algorithm can

increase congestion and therefore downgrade the Quality of Service of all users. Scheduling policies have considerable impact on multiplexing gain. The ATOM Scheduler services each stream once every 33ms (i.e. 30 fps) in a round-robin manner. Since the ATOM Data Placement Strategy ensures that the video segments are stored sequentially, round-robin is a very suitable scheduling algorithm.

4.5.5 Data Logging and User Traffic Characterization

Users access a VoD system randomly, but having detailed knowledge of access patterns (logged over long periods of time) can lead to a better managed system. For example, if the pattern shows that a video is popular on a particular Server, the system can place more copies of that video on the Server and neighbouring Servers. User access to a Server tends to be as follows: a user will decide quickly that a video is not interesting and will select a different one. However, if the user has already watched a considerable portion of the video, the likelihood of video termination before completion diminishes with time. The VoD system can commit resources on a sliding scale with higher allocation given to older streams. User access patterns to a Server will not be uniform over a given 24-hour period. Typically, the load would be moderate during the day, increasing during the evening and diminishing in the early hours of the morning. Figure 4-10 shows a hypothetical 24-hour period of access to a VoD system. Access is high in the evening, peaking at 21h00, and is moderate during the day. This type of schematic helps design schemes for resource management. It can help in updating video popularity tables, redistributing videos and reconfiguring the system during off-peak hours. Similar models can be developed for different geographical regions, video categories and individual titles. The type of video watched affects the connection holding time and use of interactive VCR commands. Consider the difference between news video clips and a full-length movie. News clips are short so the video Server should expect more variation in access patterns including the need to rewind or fast-forward or pause to look at a still image of the news event, whereas a user should generally watch a movie in its entirety. The Servers and Brokers should also have event logging functionality which specifically logs errors and system failures.

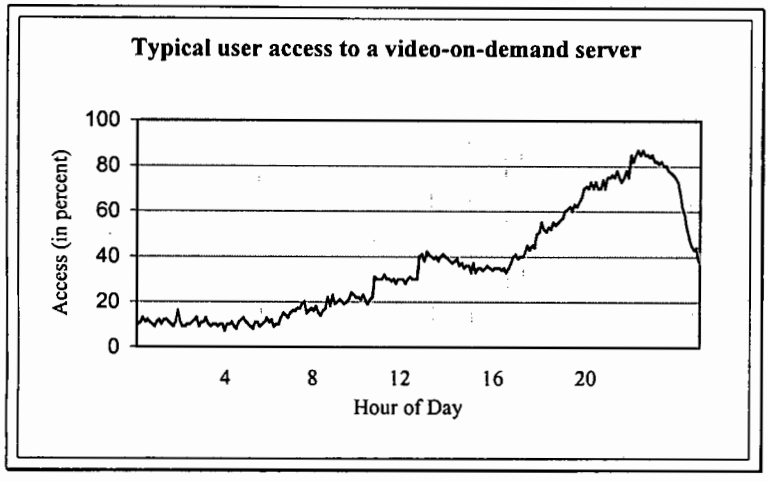

Figure 4-10 User Traffic Characterization

4.5.6 Service Contract

Chapter 2 described how Quality of Service parameters are agreed upon by the network and the user. Traditionally, the start point for the session is the Service (or Traffic) Contract. In ATOM, this form of Service Contract is not required, since the Broker is performing Connection Admission Control (as per Section 3.4.3) in which a Client is only passed the location of the Server if the Broker knows the Server can handle the new connection. It is not possible for the Client to abuse the contract since it can only play, rewind and fast-forward at the speeds designed to be handled by the Server. It is possible for the Server to abuse the contract by not providing the parameters required to transmit a particular video. The ATOM Service Contract therefore exists on the Server and consists of the parameters associated with the particular video and the B-PR file for that video. If the Server is not providing the values of the parameters in the Contract, it must rectify the situation. The Client application is unaware of the Contract, it simply knows it is paying to view the video with good quality. The design of the Service Contract draws on existing architectures [15] and the Microsoft Winsock2 Application Programming Interface specification [41]. The ATOM Service Contract parameters are presented in Table 9.

Parameter	Description
PCR	Peak Cell Rate. This parameter defines the maximum cell rate the video will ever produce. Measured in bits per second (bps).
SCR	Sustainable Cell Rate. This parameter defines an average cell rate for the full duration of the video.

BT	Burst Tolerance. Specifies how long a connection that complies to its Traffic Contract may transmit at its **PCR**
MBS	Maximum Burst Size.
CTD	Cell Transfer Delay. The Cell Transfer Delay is the total delay for a given cell from source to destination. We consider the max**CTD** and the min**CTD**.
P-P CDV	Peak-to-Peak Cell Delay Variation = max**CTD** - min**CTD**.
CDVT	Cell Delay Variation Tolerance. Defines how sensitive an application is to the variability of cell arrival for a connection.
CLR	Cell Loss Ratio. **CLR** is the ratio of the number of lost cells to the total number of cells sent by the source within a specified time interval. Typical **CLR** numbers range from 10^{-5} to 10^{-9}.
B-PR file	The B-PR file for the video enables the Server to make resource allocations according to the information about the burstiness of the video based on the amount of motion in the video.

Table 9 The ATOM Service Contract which resides on the Server for each stream

4.5.7 Connection Admission Control

Connection Admission Control in ATOM was presented in Section 3.4.3. It is a crucial part of AQOSA since it is a preventative QoS mechanism. By only allowing new connections if the Server can handle them, future congestion is minimized by not over-committing Server resources. Since the connection between Client and Server might go through more than one ATM switch it is necessary to consider what these switches must provide the stream. The ATM switches in the Internetwork must reserve the "effective bandwidth" between the **SCR** and the **PCR** and must reserve buffers proportional to the **BT**.

4.5.8 Traffic Shaping

Traffic shaping involves changing the source traffic rate to satisfy a predefined constraint. Two common uses are to reduce peak bandwidth and to reduce delay variation. Parekh [42] has shown that a combination of traffic shaping at the edge of the network and scheduling in the network can provide hard performance guarantees.

Therefore, AQOSA includes traffic shaping at the ingress to the network to help provide end-to-end service guarantees. The MPEG encoding process produces frames that vary widely in size due to the compression algorithm. This means that the video is variable bit rate including a large I-frame at the start of each GOP. Bandwidth smoothing (traffic shaping) reduces the maximum peak data transfer for variable bit rate video. The amount of shaping possible is limited by the amount of delay that can be introduced without degrading the video and audio quality. For archived video (as opposed to video from live videoconferencing), the availability of frame sizes provides a traffic profile that can be used to create a transmission schedule that consists of a sequence of fixed rates based on the traffic profile. The aim of the transmission schedule is to ensure that overflow and underflow are avoided.

A traditional traffic shaping technique that can be implemented on the ATM switch to which the Server is connected is Leaky Bucket. This implementation is popular due to its simplicity of design and the ease with which it can be implemented at the UNI. Cells arriving at a token leaky bucket are allowed through to the network if there are tokens available, taking a token out of the bucket as it is allowed through. Cells are dropped if all the tokens in the bucket have been "used up". New tokens are generated at a constant rate and placed in the bucket to allow incoming cells to go through. The "size" of the bucket, i.e. the number of tokens that it can hold, determines the number of tokens that can be transmitted back to back, and thus controls the burst size or Burst Tolerance (BT). The rate at which tokens are generated determines the cell rate to which the traffic is being shaped. A disadvantage of this technique is that it can introduce delay, degrading the quality of the video.

Vine [35] presents a summary of three bandwidth smoothing techniques for video-on-demand including bandwidth reduction comparisons. In order for the Server to reserve bandwidth for a stream, it must reserve a timeslot equal to the time needed to transmit the largest frame in the video. This is inefficient since the video may contain one massive frame with the other frames being much smaller. Clearly, smoothing of the video would eliminate this large frame, thus reducing the maximum bandwidth need of the video. The three bandwidth smoothing techniques are now analyzed.

- *Short Term Bandwidth Smoothing*

This technique is based on the principle of macroframing which means that the smoothing period is one Group of Pictures (GOP). The smoothing technique is simply an average over the macroframe interval. This creates a

constant bit rate over the macroframe interval. This is illustrated in Figure 4-11. The largest frame in the GOP is the I-frame at 11000 bytes. If the overall bit rate was dimensioned to provide for this frame, the bit rate would be

$$\frac{11000*30*8}{1024*1024} = 2.52 \text{ Mbps.}$$

However, applying macroframing results in an average frame size of 5166 bytes (i.e. 77 500 bytes/15 frames). This represents a constant bit rate of

$$\frac{5166*30*8}{1024*1024} = 1.18 \text{ Mbps.}$$

This is a marked improvement and is a decrease of 53% in peak bandwidth. The Traffic Smoothing algorithm on each ATOM Server is Short Term Bandwidth Smoothing using 1 GOP as the macroframe.

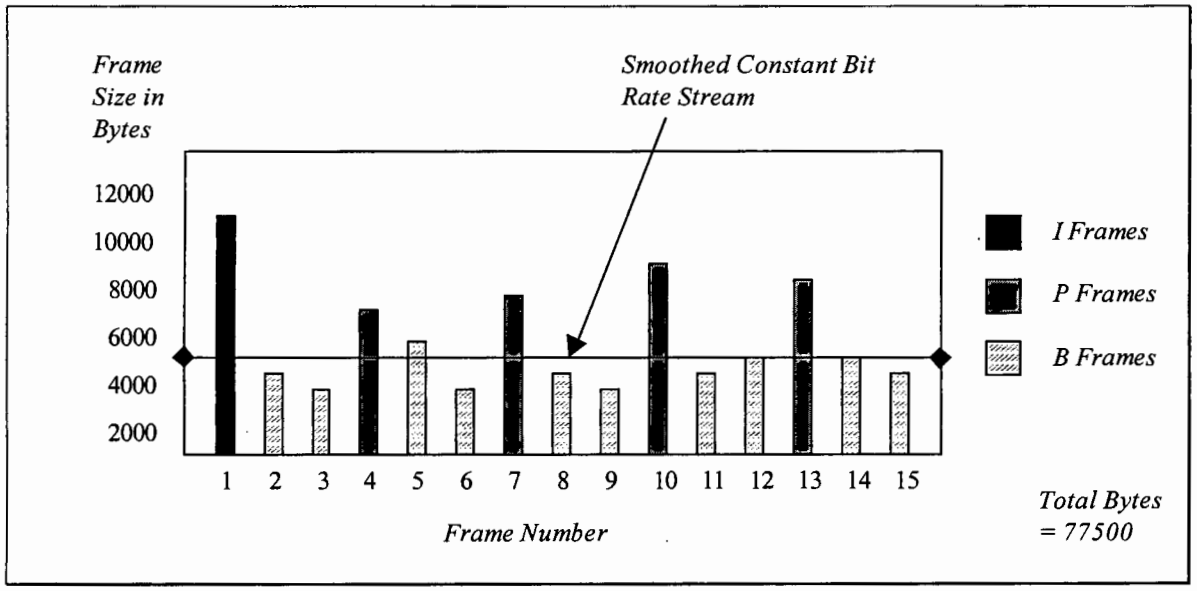

Figure 4-11 Macroframe Smoothing

Vine [35] presents the results of applying the macroframing technique to three MPEG video files. Table 10 lists these results. As an example, the largest frame in **chicken.mpg** is 33372 bytes. This creates a maximum

instantanenous burst of 7.64 Mbps. Macroframing reduces this to 1.74 Mbps. Macroframing will cause a delay due to the time needed to smooth the data. This could be avoided by delaying the playback on the Client side.

MPEG Video	Number of Frames	Largest Frame (in bytes)	Normal Maximum Bandwidth	Macroframe Maximum Bandwidth
chicken.mpg	4793	33372	7.64 Mbps	1.74 Mbps
around.mpg	750	25996	5.95 Mbps	1.69 Mbps
pikeplace.mpg	2034	28658	6.56 Mbps	1.71 Mbps

Table 10 Comparisons of video peak bit rates before and after macroframing

- *Long Term Bandwidth Smoothing*

Long Term Bandwidth Smoothing algorithms like the Critical Bandwidth Algorithm (CBA) use a graphical analogy of the transmission schedule to solve the bandwidth smoothing problem. The CBA prevents buffer underflow and overflow while minimizing the number of bandwidth increases that occur in the entire bandwidth plan. Results for the CBA are presented in Table 11 in the column CBA Maximum Bandwidth.

- *Hybrid Bandwidth Smoothing*

The next step is to consider a hybrid CBA/Macroframing technique. Vine [35] presents bandwidth reduction calculations of a hybrid CBA/Macroframing technique in Table 11 in the column CBA/Macroframe Maximum Bandwidth. Consider **chicken.mpg**. For CBA, the peak bit rate is reduced from 7.64 Mbps to 1.62 Mbps, an improvement of 78.8%. For CBA/Macroframing, the peak bit rate is reduced from 7.64 Mbps to 1.609 Mbps, an improvement of 78.9%. Although the results are good, the implementation is too complicated for ATOM. The chosen method, Short Term Bandwidth Smoothing, provides excellent gains with simple implementation.

MPEG Video	Number of Frames	CBA Maximum Bandwidth	CBA/Macroframe Maximum Bandwidth
chicken.mpg	4793	1.62 Mbps	1.609 Mbps
around.mpg	750	1.67 Mbps	1.608 Mbps

pikeplace.mpg	2034	1.72 Mbps	1.610 Mbps

Table 11 Comparison of CBA and CBA/Macroframing algorithms

4.5.9 Flow Control

Flow control attempts to control the application bit rate by keeping track of conditions on each network module in the flow. Available Bit Rate (ABR) is the only ATM service where the network provides the source with feedback on how its traffic is affecting the overall traffic load inside the network. The source can reduce its transmission rate if it is negatively affecting the overall traffic load. If the ABR mechanism is available in the network, it is worth consideration as the Traffic Class. A number of researchers have investigated the use of ABR for multimedia transport. Zheng and Atiquzzaman [43] use the ABR service to provide a peak bit rate for short bursts in order to obtain Fast Buffer Fillup at the Client machine after a rewind or fast-forward operation for a video-on-demand system. Vandalore et al [44] show that multimedia applications can be efficiently transported over the ATM ABR service. They assert that video applications can use feedback information to adjust their rates to efficiently use the available bandwidth but they also highlight areas of multimedia over ABR in which more extensive performance analysis is required. Efficient connection admission control (CAC) and usage parameter control (UPC) algorithms need to be developed and guidelines for setting ABR source parameters to provide support for point-to-multipoint connections transporting multimedia must be developed. These issues notwithstanding, flow control must form part of the ATOM QoS design as the advantages provided by feedback can improve the system.

4.5.10 Multicast

ATM supports multicast, which means transmitting packets to multiple destinations. Video conferencing is a good example of multicast, since it is necessary to send the same video and audio information to all conference participants simultaneously. The applicability of multicast to video-on-demand must be tested. An initial reaction is that it has no relevance in the wide area network (WAN) since the possibility of two (or more) Clients playing the same video at precisely the same time is remote. However, it is statistically possible for this to happen and therefore worth including in the design. The module will detect that two (or more) Clients are at the same point in a video and will add them to a group, thus having one call to the group address rather than individual calls to each Client. Of course, as soon as one Client chooses a stop, pause, rewind or fast-forward operation, the multicast module will have to remove it from the multicast group. In the LAN environment, e.g. a

networked multimedia laboratory, a video-on-demand system containing educational videos would likely see many opportunities for the multicast module to be used since there would be a limited number of videos and a high probability of students watching the same video at similar times especially if instructed to do so by a lecturer. Another issue is N-VoD (near video-on-demand). This means that the same movie starts playing at a fixed interval, e.g. every 15 minutes. Interested viewers join the next multicast session. Since ATOM provides True-VoD this is not applicable.

4.5.11 Circular Rewind / Fast-Forward Buffer

The Client needs some buffer space to smooth any fluctuations caused by network delays. At the start of the first video or after a fast-forward or rewind, this buffer needs to be filled with video data before the video can be viewed. This causes a start-up delay which is a function of the buffer size and the network bandwidth. The buffer must be correctly sized in order to avoid underflow if the network is congested. Zheng and Atiquzzaman [43] derive a formula for the minimum capacity of the Client buffer which shows that the Client buffer size is determined by the type of movie and the level of user interactivity. The numerical results provided in [43] show that typical Client buffer sizes are in the range of 200 to 650 Kbits. The 650 Kbits (i.e. 81.25 KBytes) buffer is very close to the average size of an MPEG-1 GOP (i.e. 96 KB). In fact, in order to start decoding MPEG-1, the Client only requires the first I, P and 2 B-frames which require much less than 81.25 KB. Often when the user requests a FF-RW operation, it will be localized, e.g. to replay a recent scene of the video or to scan forward for the next scene. Instead of requesting the Server to perform this operation, it is envisaged that a Client buffer could implement the rewind or fast-forward operation (assuming it has enough of the already-played or yet-to-be-played video buffered). In order to be able to provide enough frames to skip forward or back a few scenes, the ATOM Client buffer should be equal to a number of GOPs. Since each MPEG-1 GOP is 0.5 of a second of video (for 30 fps video) and has an average size of 96 KB, it is proposed to dimension the buffer to 5 seconds of video which is approximately 1 MB of space. This will give a window of 5 seconds of rewind or fast-forward in which the Server does not have to provide the data. This idea is similar to the VCR-Window presented by Feng [45]. Assume a circular buffer as in Figure 4-12. The buffer has a point-of-transmission (POT) and a point-of-play (POP). The Rewind area is the amount of data that has been played back and is still in the buffer (i.e. it has not been overwritten by the POT). The Fast-Forward area is the data that has not yet been played back (i.e. it has not been reached by the POP).

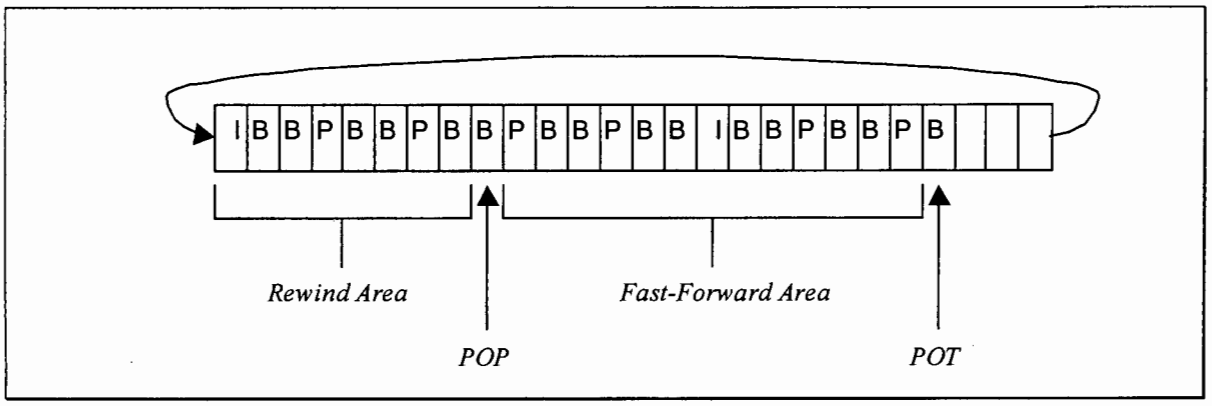

Figure 4-12 The Circular Rewind / Fast-Forward Buffer

Under normal conditions the POP will always lag the POT but under network congestion conditions the POP could pass the POT since no data will be available. This is called buffer underflow. If the POT passes the POP we have buffer overflow since the system is too slow in playing back the data in the buffer. The Rewind and Fast-Forward Areas are useful because they can be used for a short rewind or fast-forward operation without having to tell the Server to provide for these tasks. The Client will inform the Server to pause transmission and then the Rewind or Fast-Forward area will be used for the operation. No renegotiation of network bandwidth is required since the data is already present in the Client buffer. The Server will only be told to transmit again when (a) the playback from the Rewind Area reaches the POP, (b) the rewind extends further than the Rewind Area, or (c) the playback from the Fast-Forward Area reaches the POT.

4.5.12 Usage Parameter Control for QoS Monitoring and Policing

The Service Contract QoS parameters are considered to be the Conformance Definition by which the Server can police the connection using Usage Parameter Control (UPC). As stated in Section 4.5.6, the ATOM Service Contract exists to enable the Server to monitor itself, since it is not possible for the Client to consume bandwidth higher than any predefined playback rates. So the Server will monitor the parameters in the stream and if it discovers that they are less than required, it must rectify the situation. A typical UPC implementation is the Generic Cell Rate Algorithm (GCRA) which defines cell by cell conformance to a generic rate and includes a certain tolerance. For ATOM, there are two GCRAs operating. For the first, the policed rate would be the Peak Cell Rate (PCR) and the tolerance would be the Cell Delay Variation Tolerance (CDVT). For the second, the policed rate would be the Sustainable Cell Rate (SCR) and the tolerance would be the Burst Tolerance (BT).

Chapter 5 Software Design and Implementation and ATOM Applications

5.1 Introduction

This Chapter presents a design for fast-forward and rewind of a video stream called the ATOM FF-RW Method and describes the object-oriented software implementation of ATOM. First, four algorithms for fast-forward and rewind are investigated in Sections 5.2.1.1 to 5.2.1.4 by showing comparisons for a specific MPEG-1 file. The limitations of these methods are presented. The ATOM FF-RW method is presented in Section 5.2.2 and applied to the same MPEG-1 file. A significant decrease in bandwidth is obtained, and the resultant stream is very constant, reducing the chance that the stream will add to network congestion. The object-oriented design of the ATOM System is then presented and the classes of the Server, Broker and Client are detailed in Section 5.3. For each class, the included header files, the external objects and the member functions are listed. A description of each class is given, emphasizing the interaction between classes. Section 5.4 investigates possible ATOM applications, namely ATOM in the Virtual Private Network in Section 5.4.1 and ATOM in the multimedia teaching laboratory in Section 5.4.2.

5.2 Fast-forward and rewind

There are a number of techniques proposed in the literature for implementing fast-forward and rewind (FF-RW) of the video streams. However, very few actual implementations exist. First, a number of alternatives from the literature are presented and then the ATOM FF-RW technique is detailed. The most effective way of presenting these algorithms is to show comparisons for a specific MPEG-1 file. The video chosen for this comparison is **porsche.mpg**, which is shown in Figure 5-1 and detailed in Table 12. The sample video has a large number of scene changes over the 10.6-second duration. The large I-frames are clearly visible as spikes in Figure 5-1.

Note: The values in Figures 5-1 to 5-5 and Tables 12 to 15 were obtained as output from software written for this task called Toolbox, coded by the Author. Source code for Toolbox is available in Appendix A and on the accompanying CD.

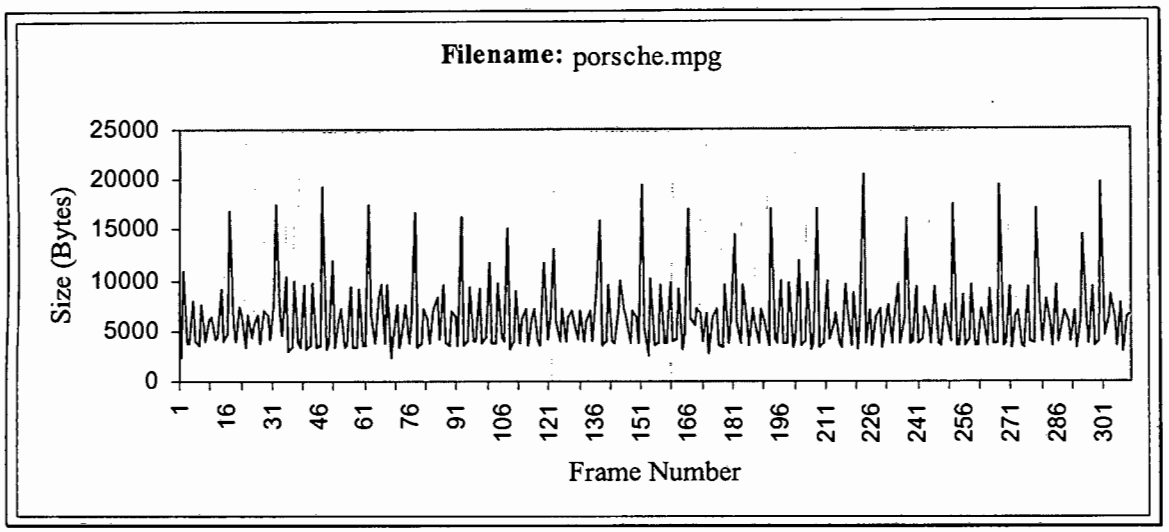

Figure 5-1 The sample video porsche.mpg used for comparisons of FF-RW techniques

Size (in Bytes)	1966082
Number of I-frames	22
Number of B-frames	198
Number of P-frames	79
Total other frames	10
Total Number of frames	309
Running Time (in seconds)	10.6
Frame Rate (in frames per second)	30
Average Bit Rate (Mbps)	1.42

Table 12 Detailed description of porsche.mpg

5.2.1 Alternative methods for FF-RW from the literature

5.2.1.1 Increased Playback Rate

This fast-forward method from Dey-Sircar et al [46] simply assumes that the Server is able to stream the video at a faster rate. For example, assume the stream normally plays at 15 frames per second (fps). With the Increased Playback Rate method, the stream will play at 30 fps. This is only twice the normal playback rate. The obvious problem with this method is that it increases the load on both the Server and network. Also, MPEG decoders aren't designed to run at more than 30 fps so the percentage speed-up possible is very small. With the sample video **porsche.mpg**, no speedup is possible.

5.2.1.2 Skip Groups of Pictures (GOPs)

This method maintains the same playback rate but skips entire groups of frames. For example, to achieve a speed-up of three, every third GOP is sent by the Server and consumed by the Client. The disadvantage is that the visual playback does not look faster, but it may be acceptable since the user is searching by eye for a particular part of the video and would appreciate no speedup of displayed video. Figure 5-2 and Table 13 present the new video stream. The I-frame peaks are still clearly visible.

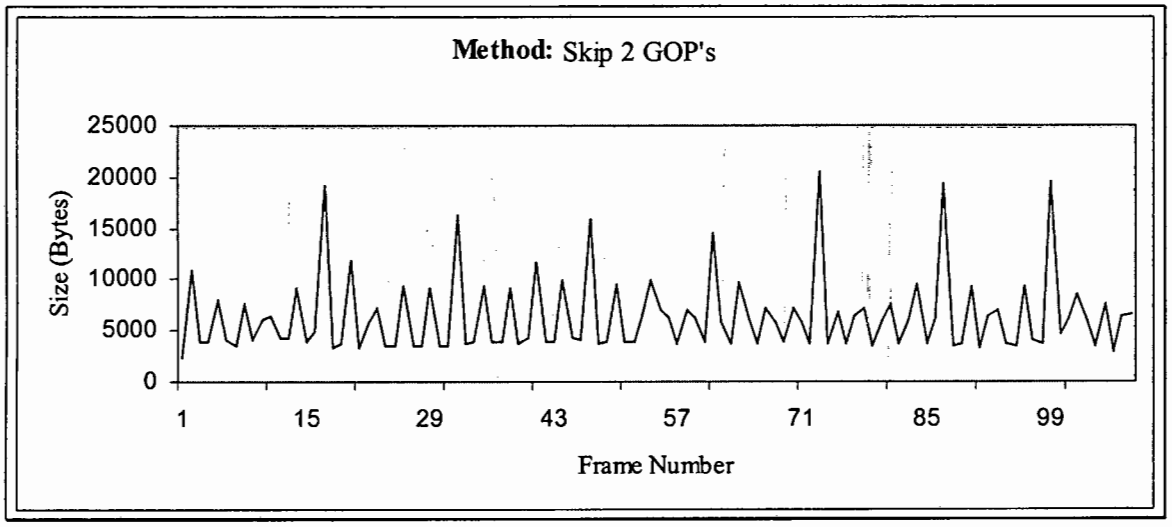

Figure 5-2 Method: Skip 2 GOP's

Size (in Bytes)	686653
Number of I-frames	8
Number of B-frames	64
Number of P-frames	26
Total other frames	10
Total Number of frames	108
Running Time (in seconds)	~ 3.6
Frame Rate (in frames per second)	30
Average Bit Rate (Mbps)	1.45

Table 13 Skip 2 GOP's

5.2.1.3 Skip Independent Sequences

This method from Ozden et al [47] maintains the same playback rate but skips parts of each GOP. In this method the first N frames of each GOP starting at the I-frame and ending with the first P-frame are delivered, skipping all frames of the GOP after the first P-frame. For example, in a GOP of IBBPBBPBBP, only the first 4 frames, IBBP are delivered. This method is better visually than skipping entire GOPs. Figure 5-3 and Table 14 show this method applied to **porsche.mpg** with N = 4. As can be seen in Figure 5-3, the bandwidth of the stream increases substantially. There are now 8 I-frames every second where before there used to be 2.

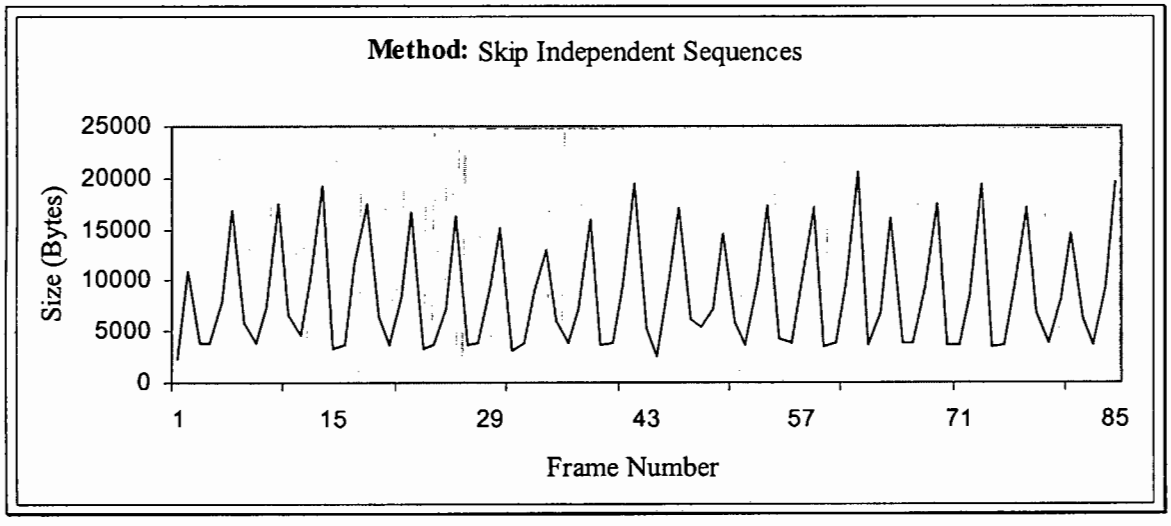

Figure 5-3 Method: Skip Independent Sequences

Size (in Bytes)	731509
Number of I-frames	22
Number of B-frames	40
Number of P-frames	21
Total other frames	1
Total Number of frames	84
Running Time (in seconds)	~ 2.8
Frame Rate (in frames per second)	30
Average Bit Rate (Mbps)	1.99

Table 14 Skip Independent Sequences

5.2.1.4 Alternative File

A special file is created for use in FF-RW. This does not increase the Server and network load but requires extra storage space on the Server. An example of this is presented by the Stony Brook Video Server Project [SBV99]. They propose the use of a separately encoded, lower resolution version of the original video to support FF-RW encoded at a lower frame rate (e.g. 10 fps as opposed to 30 fps). During fast-forward, the alternative file is sent at the same bit rate (not frame rate) as the normal video, thus increasing the frame rate. For example, consider a video encoded at 30 fps with an average bit rate of 1.5 Mbps. Assume the alternative file is encoded at 10 fps with an average bit rate of 0.5 Mbps. When the alternative file is played at 1.5 Mbps, it is essentially three times faster.

Another method is to store three versions of a video namely (a) normal, (b) fast-forward and (c) rewind. The fast-forward disk has a certain number of frames skipped periodically. The rewind disk also has skipped frames but the frames are stored in reverse order.

5.2.2 The ATOM FF-RW Method

The design criteria are as follows:

- Provide a fast response to user requests. When a FF/RW/Pause request is selected the system should activate these controls immediately.
- When play resumes after a FF or RW, the closest frame in the video must be transmitted to the user in the shortest amount of time as the new starting point.

When the user selects fast-forward or rewind, the Server switches to an alternative playback mode in which only I-frames are sent to the Client. During normal playback of 30 fps, 2 I-frames are sent every second. To achieve a speedup, more than 2 I-frames are sent in this period. For example, to achieve a speedup of 3 times normal playback, 6 I-frames are sent every second. Since I-frames are large this could increase the bandwidth requirements to unacceptable levels. The ATOM FF-RW Method therefore averages the I-frame data over the one second period into 30 blocks of data. The resulting bandwidth is significantly lower than the original bandwidth. Figure 5-4 depicts the ATOM FF-RW method applied to **porsche.mpg** for a playback speed of 3 times normal speed (i.e. 6 I-frames are averaged and sent per second). Figure 5-5 plots this ATOM FF-RW method stream against the normal stream. Table 15 presents the average bit rate for this ATOM FF-RW stream. Apart from the significant decrease in bandwidth, the resultant stream is very constant, reducing the chance that the stream will add to network congestion.

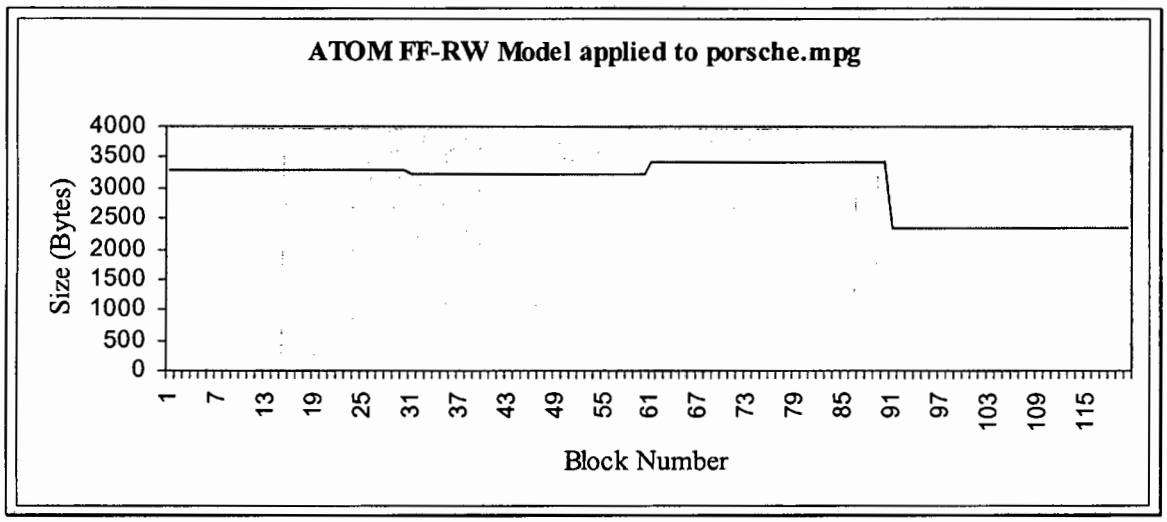

Figure 5-4 ATOM FF-RW Model applied to porsche.mpg

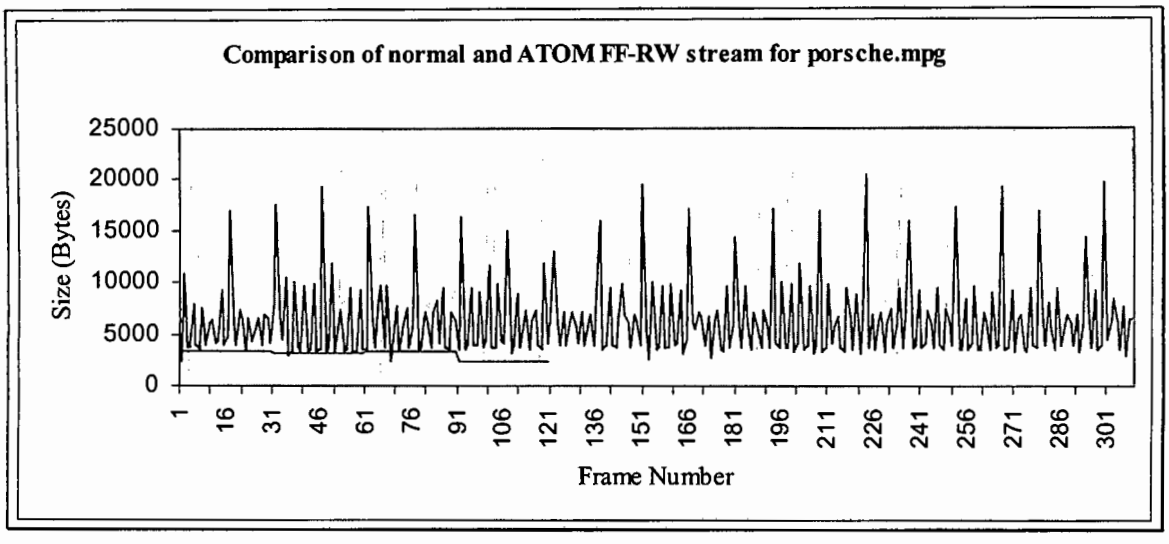

Figure 5-5 Comparison of ATOM FF-RW stream and normal stream

Size (in Bytes)	367920
Number of averaged "frames"	120
Number of B-frames	0
Number of P-frames	0
Total other frames	0
Total Number of averaged "frames"	120
Running Time (in seconds)	~ 4
Frame Rate (in frames per second)	30
Average Bit Rate (Mbps)	0.7

Table 15 ATOM FF-RW stream

5.2.2.1 Start position after rewind or fast-forward operation

When play is to resume after a rewind or fast-forward operation, the data placement strategy (DPS) will locate the I-frame which is closest to the point in the video chosen by the user. The Server must begin transmission from this point in the video in the shortest amount of time. The selected I-frame may not be the exact frame in

the video when the user stopped the rewind or fast-forward operation (e.g. a B or a P-frame) but will be within one GOP. This represents an error of 0.5 seconds (assuming MPEG-1 video at 30 fps) and is tolerable.

5.3 Software Module Overview

ATOM consists of Servers, Brokers and Clients. Each software entity has C++ classes which attempt to model the real world. Figures 5-6, 5-7 and 5-8 depict the classes and real world equivalent for the ATOM Server, Broker and Client. The classes are described in Sections 5.3.1, 5.3.2 and 5.3.3. The source code is provided in Appendix A and on the accompanying CD and is fully commented.

The software was implemented in Microsoft Visual C++. The networking stack (or Service Provider Interface) is provided by FORE, the manufacturer of the ATM network adapters. A setup program is run which registers the FORE Service Provider Interface with Winsock 2. When a socket is created, the requirement for an ATM network adapter is passed in as a parameter. The Winsock 2 library recognises the FORE Service Provider Interface and any subsequent socket calls are handled by the FORE stack.

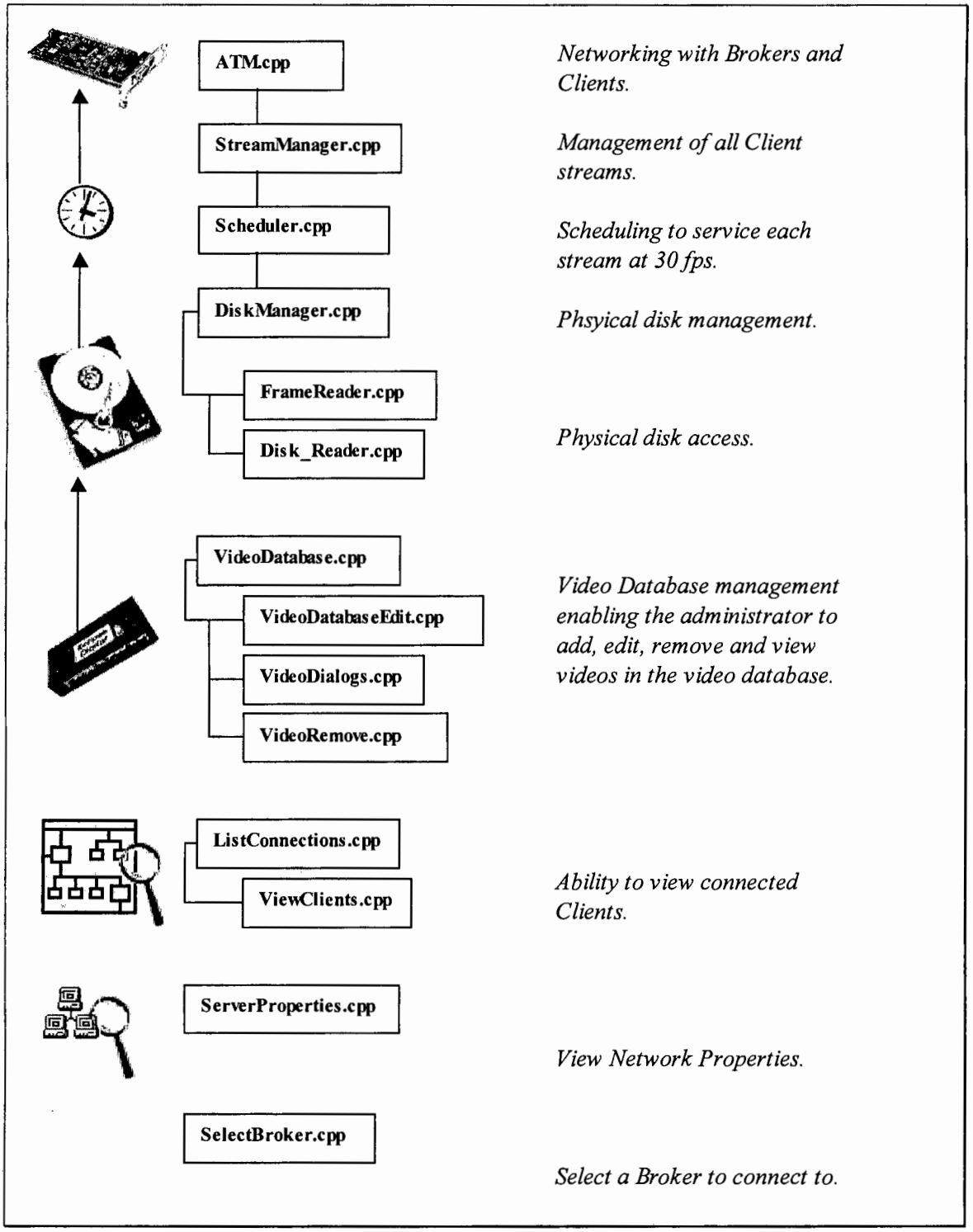

Figure 5-6 Classes for the ATOM Server

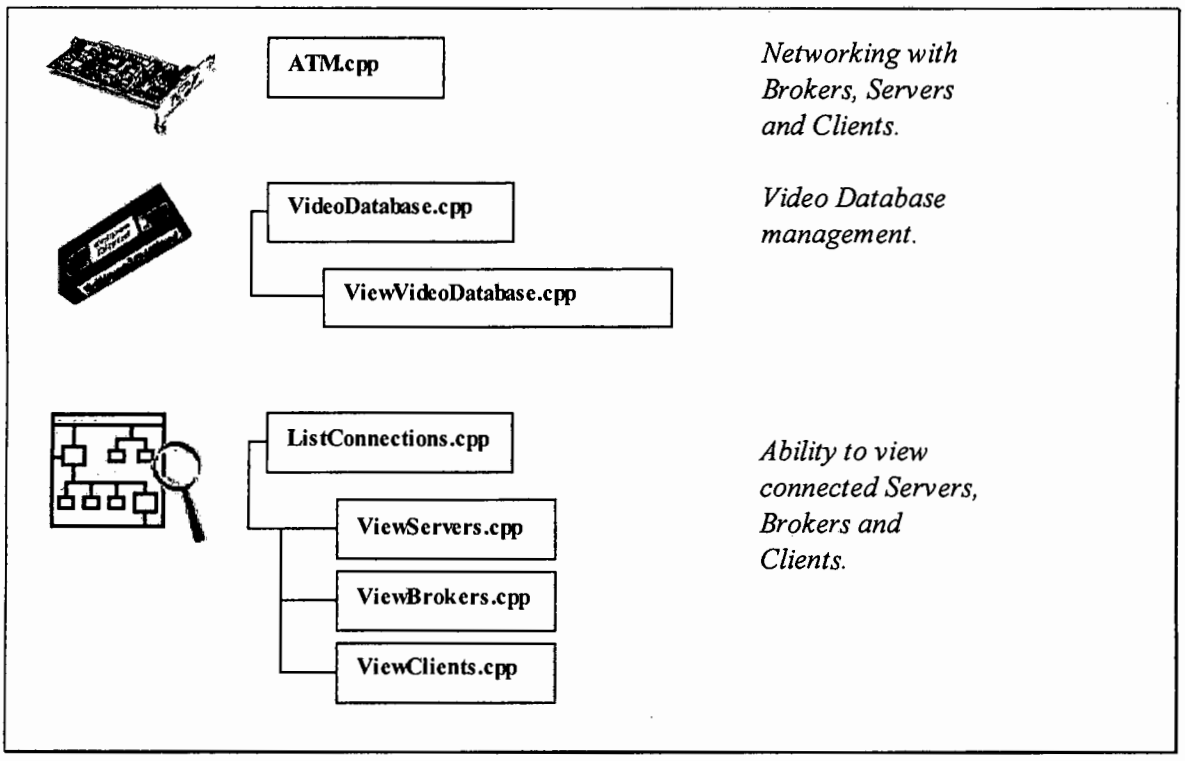

Figure 5-7 Classes for ATOM Broker

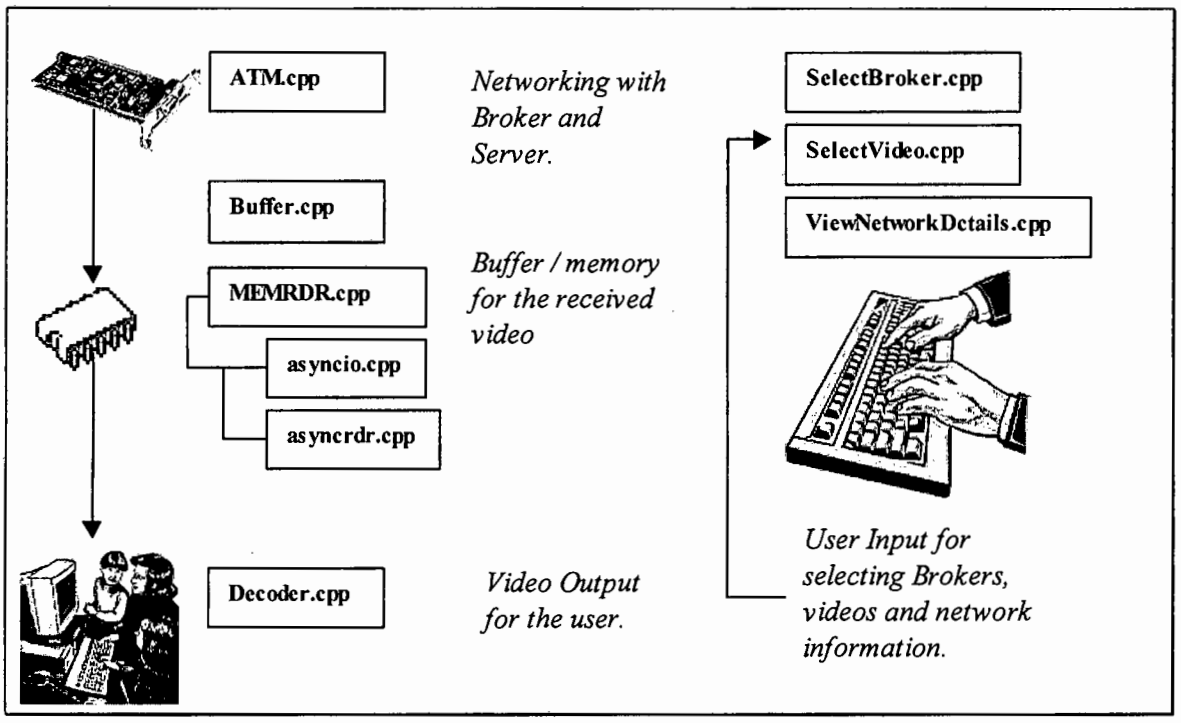

Figure 5-8 Classes for ATOM Client

5.3.1 Classes for ATOM Server

- **ATM.cpp**

Header Files:	*Stream_Manager.h, VideoDatabase.h, ListConnections.h, winsock2.h, ws2atm.h, ws2spi.h, ATM.h*
External Objects:	*VideoDatabase Video, CStream_Manager SM, ListConnections Clients*
Member Functions:	*Launch(), Close_Winsock(), ListenToConnectionSocket(), ClientComms(), GetBinary(), TxSection(), SendFinished(), Receive_Movie_Choice(), NewBufMax(), prepare_struct(), Get_Name_By_Address(), Read_File(), Parse_Line(), Get_Hex(), get_atm_address(), Setup_View_Movies()*

This class implements the ATM network interface. The Server creates a socket (*Broker_socket*) and connects to a Broker, sending its video database. The Server then creates a socket (*SpawnSocket*) to listen for Client connection requests. This socket is bound to the ATM address of the Server and a listen queue is started. Whenever a new Client is accepted a new socket is created and a thread is spawned to deal with that Client for the duration of the connection. Communication between the Client and Server is handled by the ClientComms() member function. The ClientComms() function creates a second socket for video data streaming and connects to the Client on this socket. The first socket is used for control information between the Client and Server. The TxSection() member function is used by the Scheduler.cpp class to send the video data. It takes in blocks of data and a Client's socket ID and then sends the data over that socket.

- **StreamManager.cpp**

Header Files:	*Stream_Manager.h, ListConnections.h, Scheduler.h, Disk_Manager.h, ATM.h*
External Objects:	*CATM ATM, ListConnections Clients*
Objects Created:	*CScheduler Sched, CDisk_Manager DM*
Member Functions:	*InitSM(), StartDM(),StartTimer(), StartSchedule(), DoSM(), NewMovie(), PlayMovie(), StopMovie(), CancelMovie(), CloseSM()*

This class manages each Client connected to the Server. It is responsible for setting up and removing streams from the Scheduler and Disk_Manager objects. It creates a CScheduler object and a CDisk_Manager object. It spawns three threads: a timer thread, a disk reading thread and a scheduling thread. It is responsible for handling

user inputs while the Client is in playback mode. This means it will handle all fast-forward, stop and pause type requests and tell the relevant objects what to do. The ATM.cpp class calls the member function DoSM() whenever a control character is received from the Client. The control characters are: "N" - new movie, "S" - stop, "P" - pause, "C" - cancel movie. If a new movie is requested, buffers are created for the stream. When play is requested, the stream is added to the Scheduler object which will then service the stream with video every 33ms (i.e. 30 frames per second).

- **Scheduler.cpp**

Header Files: *Scheduler.h, Disk_Manager.h, ATM.h, mmsystem.h, math.h, Globals.h*
External Objects: *CATM ATM*
Member Functions: *Timer(), DoSchedule(), GetBuffer(), SetUpBuffers(), SetMovieActive(), SetMovieActive(), RemoveBuffers(), SetQuit(), SetBufMax(), SetDM()*

This class provides the scheduling function to service each stream with video every 33ms, thus ensuring that each Client receives 30 frames per second. An array of SCHEDULER_INFO structures holds the connection details of each stream. The SCHEDULER_INFO structure is defined in Globals.h and provides two buffers for each stream.

- **Disk_Manager.cpp**

Header Files: *FrameReader.h, DiskReader.h*
Member Functions: *GetFrame(), GetMacroFrame(), StartDisk(), OpenMEGFile(), CloseMPEGFile(), ResetMPEGFile(), GetBlock(), SkipStuffBytes(), MoveOffCurrentStartCode(), FindNextStartCode(), SetEndOfMPEGBuffer(), SetFileFinished(), GetAudioFrame()*

Disk_Manager.cpp accesses the physical disk to retrieve segments of video. It uses a read-ahead process to ensure that enough video is available for transmission.

- **VideoDatabase.cpp**

Header Files: *VideoDatabase.h*

Member Functions: *Search_On_Video_Name(), Get_Server_Address(), Search_on_Video_Type(), Delete_by_Video_Name(), Insert_In_Order(), Get_Next(), Set_Current(), Empty(), Get_Field(), Save(), Load(), Parse_Video_String(), Create_Video_List_For_Broker(), Initialize_List()*

This class implements a linked list for a video database. This video database is sent to the Broker during registration. The Server administrator can add, remove or view the database.

- **VideoDatabaseEdit.cpp**

Header Files: *VideoDatabaseEdit.h, VideoDatabase.h*
External Objects: *VideoDatabase Videos*
Member Functions: *OnInitDialog(), OnOk(), OnCancel(), OnSelchangeList1()*

This class implements a dialog box to enable the Server administrator to edit entries in the video database. It uses the Videos object created in the VideoDatabase class.

- **VideoDialogs.cpp**

Header Files: *VideoDialogs.h, VideoDatabase.h*
Member Functions: *OnInitDialog(), OnOk(), OnCancel()*

This class allows the Server administrator to add a new video to the video database.

- **VideoRemove.cpp**

Header Files: *VideoRemove.h, VideoDatabase.h*
External Objects: *VideoDatabase Videos*
Member Functions: *OnInitDialog(), OnOk(),OnCancel()*

This class implements a dialog box to enable the Server administrator to remove entries from the video database. It uses the Videos object created in the VideoDatabase class.

- **ListConnections.cpp**

Header Files: *ListConnections.h*

Member Functions: *Search_on_Name(), Delete_by_Name(), Insert_In_Order(), Set_Current(), Empty(), Get_Next(), Update_Video_by_Bufnum(),Get_Field(), Initialize_List()*

This class implements a linked list which is used to track connected Clients.

- **ViewClients.cpp**

Header Files: *ViewClients.h, Globals.h, ListConnections.h*

External Objects: *ListConnections Clients*

Member Functions: *OnInitDialog(), OnOk(), OnCancel(),OnSelchangeList1()*

This class implements a dialog box to enable the Server administrator to view network details of all connected Clients. It uses the Clients object created in the ListConnections class.

- **ServerProperties.cpp**

Header Files: *ServerProperties.h, atm.h*

External Objects: *CATM ATM*

Member Functions: *OnInitDialog(), OnOk()*

This class implements a dialog box to enable the Server administrator to view network details of the Server. It uses the ATM object created in the CATM class.

- **SelectBroker.cpp**

Header Files: *SelectBroker.h*

Member Functions: *OnInitDialog(), OnOk(), OnCancel()*

This class implements a dialog box to enable the Server administrator to select and connect to a Broker.

5.3.2 Classes for ATOM Broker

- **ATM.cpp**

Header Files: *ATM.h, VideoDatabase.h, ListConnections.h, winsock2.h, ws2atm.h, ws2spi.h, Globals.h*

External Objects: *VideoDatabase Video_List, ListConnections Clients, ListConnections Servers, ListConnections Brokers*

Member Functions: *Launch(), Close_Winsock(), ListenToConnectionSocket(), ClientComms(), QOS_Engine(), BrokerComms(), ServerComms(), Send_Videos_To_Client() GetBinary()*

This class implements the ATM network interface. The Broker creates a socket (*SpawnSocket*) to listen for Server, Broker and Client connection requests. The socket is bound to the ATM address of the Broker and a listen queue is started. Whenever a new Server, Broker or Client is accepted a new socket is created and a thread is spawned to deal with the Server, Broker or Client for the duration of the connection. Communication is handled by the ServerComms(), BrokerComms() and ClientComms() member functions. The QOS_Engine() member function acts as a Connection Admission Control function. It negotiates quality of service and then passes the ATM address of the Server to the Client.

- **VideoDatabase.cpp**

Header Files: *VideoDatabase.h*

Member Functions: *Search_On_Video_Name(), Get_Server_Address(), Search_on_Video_Type(), Delete_by_Video_Name(), Insert_In_Order(), Get_Next(), Set_Current(), Empty(), Get_Field(), Save(), Load(),Parse_Video_String(), Create_Video_List_For_Client(), Initialize_List(),Delete_by_Server()*

This class implements a linked list for a video database. This video database is sent to the Client as requested. The Broker merges the video databases of its Servers and Peer Brokers, maximising the Client's ability to find the desired video.

- **ViewVideoDatabase.cpp**

Header Files: *ViewVideoDatabase.h, VideoDatabase.h*

External Objects: *VideoDatabase Video_List*

Member Functions: *OnInitDialog(), OnOk(), OnSelChangeList1()*

This class allows the Broker administrator to view the video database.

- **ListConnections.cpp**

Header Files: *ListConnections.h*

Member Functions: *Search_on_Name(), Delete_by_Name(), Insert_In_Order(),Set_Current(), Empty(), Get_Next(), Update_Video_by_Bufnum(),Get_Field(), Initialize_List()*

This class implements a linked list which is used to track connected Servers, Brokers and Clients.

- **ViewServers.cpp**

Header Files: *ViewServers.h, ListConnections.h*

External Objects: *ListConnections Servers*

Member Functions: *OnInitDialog(), OnOk(),OnSelchangeList1()*

This class implements a dialog box to enable the Broker administrator to view network details of all connected Servers. It uses the Servers object created in the ListConnections class.

- **ViewBrokers.cpp**

Header Files: *ViewBrokers.h, ListConnections.h*

External Objects: *ListConnections Brokers*

Member Functions: *OnInitDialog(), OnOk() ,OnSelchangeList2()*

This class implements a dialog box to enable the Broker administrator to view network details of all connected Peer Brokers. It uses the Brokers object created in the ListConnections class.

- **ViewClients.cpp**

Header Files: *ViewClients.h, Globals.h, ListConnections.h*
External Objects: *ListConnections Clients*
Member Functions: *OnInitDialog(), OnOk(), OnSelchangeList1()*

This class implements a dialog box to enable the Broker administrator to view network details of all connected Clients. It uses the Clients object created in the ListConnections class.

5.3.3 Classes for ATOM Client

- **ATM.cpp**

Header Files: *ATM.h, VideoDatabase.h, SelectVideo.h, winsock2.h, ws2atm.h, Buffer.h, uuid.h*
External Objects: *CBuffer Buffer, CStringArray vids_name, CStringArray vids_category CStringArray vids_time, CStringArray vids_description*
Member Functions: *Init_Winsock(), Close_Winsock(), GetBinary(), Get_Hex(), GetVideo(), SetBuffer(), Negotiate_QOS(), Get_Videos(), Send_Control(), Send_Broker(), Get_Name_By_Address(), Parse_Line(), Read_File(), get_atm_address(), Server_Connect(), Broker_Connect()*

This class implements the ATM network interface. The Client creates three sockets (*Broker_socket, control_socket, setup_video_socket*). The user selects a Broker and connects to it. The Broker sends the videodatabase. The user selects a video, negotiates QoS and is given the ATM address of the Server. The Client connects to the Server and is able to view the video with full interactive controls.

- **Buffer.cpp**

Header Files: *Buffer.h*
Member Functions: *ResetBuffer(), WriteBuffer(), ResetBuffer(), RemoveBuffer(), SetUpBuffer(), ReadBuffer(), SizeOfBuffer()*

This class implements a buffer for the video received from the Server.

- **MEMRDR.cpp, asyncio.cpp and asynrdr.cpp**

These classes implement memory reading functions required by the Decoder.cpp class.

- **Decoder.cpp**

Header Files: *memrdr.h, decoder.h, buffer.h, streams.h*

Member Functions: *SetPaused(), SetRun(), GetPropertyPage(),Decode(), ResetWin()*
PlayFileWait(), SelectAndRender()

This class renders the video data to the screen using DirectShow.

- **SelectBroker.cpp**

Header Files: *SelectBroker.h*

Member Functions: *OnInitDialog(), OnOk(), OnCancel()*

This class implements a dialog box to enable the user to select and connect to a Broker.

- **SelectVideo.cpp**

Header Files: *SelectVideo.h*

Member Functions: *OnInitDialog(), OnOk(), OnCancel(), OnButton1() ,OnSelchangeList1()*

This class implements a dialog box to enable the user to select a video.

- **ViewNetworkDetails**

Header Files: *ViewNetworkDetails.h, atm.h*

External Objects: *CATM ATM*

Member Functions: *OnInitDialog(), OnOk(), OnDoubleclickedOk()*

This class implements a dialog box to enable the user to view the network connectivity.

5.4 Applications of the ATOM System

5.4.1 ATOM in the Virtual Private Network

A Virtual Private Network (VPN) is a private networking service (normally for a company) over a public network infrastructure (spanning local and international boundaries). A VPN gives a company secure data transfer over public or shared networks and increases corporate productivity and efficiency. The company uses a large and established network but only pays for the portion of the network it uses and does not pay for the initial capital outlay. A VPN could link the global offices of a company (e.g. London, Johannesburg, New York) and allow users in those locations to browse company information and intranets in all the offices.

A company could use ATOM Servers to store videos of training, shareholders speeches, company advertisements and other company material. ATOM Client software on employees machines can access ATOM Brokers and find desired videos. Servers in each global office could store both general company video material and videos relevant to that region and country. In this way, ATOM is used to provide a company-wide video-on-demand system for all employees.

5.4.2 ATOM in the multimedia teaching laboratory

A multimedia laboratory contains PCs linked to a video Server for playback of training videos and coursework. The laboratory could be serving school students, tertiary students (university, technikon and college) or company employees. The PCs would run ATOM Client software and a Broker and Server would exist on one or more multimedia machines. The multicast function would be used when all students were watching the same video, else each student could watch a different video at his/her convenience.

Chapter 6 Conclusions and Recommendations for Future Work

Having designed and implemented the ATOM model, a number of conclusions can be drawn which highlight the fundamental decisions and results of the project. A number of recommendations for future work on this topic are highlighted.

6.1 Conclusions

1. Digital video applications require high bandwidth, low error, low delay networks.

Each digital video on a Server has massive storage and bandwidth requirements as well as strict bounds on delay and error. The large storage requirement means that dedicated Servers with large disk capacity are required to host the hundreds or thousands of videos in such a system. Each Server should stripe the videos over a disk array in order to increase overall disk bandwidth and fault tolerance. ATM networks provide the high bandwidth, low error and low delay required to satisfy the requirements of digital video. In ATOM, the video source is MPEG-1, which has an acceptable end-to-end delay of 100 ms, an acceptable bit error rate of 10^{-10} and acceptable cell loss ratio of 10^{-12}. The ATOM protocol also supports Quality of Service agreements which aid in resource allocation through connection admission control.

2. A video-on-demand system to support large Client volumes must be distributed, not centralized.

Centralised video-on-demand systems provide one Server to which all Clients must connect. In order to support thousands of simultaneous Clients, a distributed system is required with many Servers available for Client access. Intelligent placement of videos on these Servers and communication between these Servers is required to maximize the Client's chance of viewing a desired video. A well-designed distributed topology will reduce the load experienced by each Server by offering the Client alternative video Servers on which to locate a video. Distributed systems facilitate scalability through the ability to easily add more Servers. Another entity (called a Broker) can be used to collate information about the distributed system and thus provide a central repository for

Server information. Clients can perform a search of all the videos on every Server simply by connecting to a Broker.

3. Control and Operation (streaming) must be separated.

The efforts of the TINA Consortium (TINA-C), DSM-CC (Digital Storage Media - Command and Control) and DAVIC (Digital Audio and Video Council) have shown that management of streams and transport of streams are separate entities. The concept of separating the control and management of the streams from the actual transport of the streams has been adhered to in ATOM, which maintains separate virtual circuit connections for control and transport data. Control in ATOM consists of user operations like play, fast-forward, rewind and stop and tasks performed on the Broker – e.g. search and admission control.

4. The number of ATM Switched Virtual Circuits (SVCs) must be minimized

An important performance parameter in ATM is the time overhead associated with the setting up of Switched Virtual Circuits (SVCs). The FORE ASX200BX switch has an average SVC setup time of 10 ms, but as the number of connections (or hops) between switches increases, the setup time increases rapidly with the setup time for each hop taking between 45.5 ms and 100 ms. The ATOM Model minimizes the number of connections and therefore minimizes the overhead associated with SVC setup. A video-on-demand system consists of SVCs (not Permanent Virtual Circuits) since Client/Server connections are relatively short lived (in the order of minutes and hours).

5. The increased connections caused by the Broker mesh is justified by the distributed information gain.

Brokers must register with each other to form a mesh. These Brokers are then called Peer Brokers which must share information contained in their video databases. The connections between Brokers are ATM Permanent Virtual Circuits (PVCs). The mesh is fault-tolerant since if a Broker fails, all remaining Brokers are still connected to each other. The mesh topology increases the number of links required (as opposed to a bus connection for the Brokers) but the increase is tolerable since the total number of Brokers is always very much smaller than the number of Clients and Servers and the gain in distributed information greatly outweighs the extra connections. The mesh topology's most important feature is that it scales well: the number of links increases geometrically with the number of nodes in the system.

6. A Quality of Service solution must address end-to-end issues.

Most research into providing QoS guarantees in multimedia communications has focused on network-oriented traffic models and service scheduling disciplines. These guarantees are not end-to-end, they only preserve guarantees between the end-system and the network access point. QoS architectures must cover all parts of the video stream including the application, the operating system, the communication stack and the network.

6.2 Recommendations for Future Work

1. Develop a world wide web (WWW) front-end for Client access to Brokers

With the prevalence of the Internet, it is highly advisable that a website devoted to a particular ATOM implementation be developed. One particular Broker will provide a Web Server with a well-known address (e.g. www.atom-vod.co.za) from which Clients can receive the ATM address of the Broker nearest to them in terms of number of ATM hops. Since the Web Server is hosted on a Broker, the Web Server will have an up-to-date list of all connected Brokers due to the Peer Broker protocol. The Web Server could also supply software upgrades for Clients. The Client would cache information received from a visit to the Web Server, including Broker addresses. This would reduce the number of visits required to the Web Server.

***2.* Deploy the system in a wide area ATM network**

As more nodes enter the pilot ATM network in South Africa, the network will at some stage be considered wide area. ATOM should be deployed in this wide area and extensively tested. The tests should include stress testing of Servers by connecting the maximum number of Clients simultaneously and testing the Broker protocol by forcing failure of a Broker and ensuring that its Clients and Servers are redeployed correctly.

3. Develop a proprietary file format for striping

It was shown in Section 4.5.2 that a disk array provides increased overall disk bandwidth since retrieval operations can be performed in parallel. It was also shown that videos should be striped across the disk array in segments of 256 KB for optimum performance. However, Windows NT File System only supports strip sizes of

64 KB. Therefore, a file format which can support 256 KB strips and still be accessed by the Windows NT operating system RAID Level 0 controller should be developed for optimum disk bandwidth.

References

[1] NTT Software Corporation's Interspace at http://www.is.ntts.com/ispace.html

[2] Microsoft's NetShow Theater Server at http://www.microsoft.com/Theater/sitemap.htm

[3] The Centre for Distance Education, University of Texas at Arlington at http://distance.uta.edu

[4] Moskowitz, R "What are Clients and Servers anyway?" Network Computing, Client/Server Supplement, May 1993

[5] Minoli, D; Schmidt, A "Client/Server Applications on ATM Networks" Manning Publications Co. 1997

[6] Onvural, R "Asynchronous Transfer Mode Networks: Performance Issues Second Edition" Artech House Inc. 1995

[7] Tatipamula, M; Khasnabish, B; Editors "Multimedia Communications Networks - Technologies and Services" Artech House 1998

[8] Ozer, J "Video Compression for Multimedia" Academic Press, Inc. 1995.

[9] The ATM Forum Technical Committee. "Audiovisual Multimedia Services: Video-on-Demand Specification 1.1, March, 1997" available via anonymous ftp at ftp://ftp.atmforum.com/pub/approved-specs/af-saa-0049.001.pdf

[10] ISO/IEC JTC1/SC29/WG11 "Information Technology - Generic Coding of Moving Pictures and Associated Audio: Digital Storage Media - Command and Control" ISO/IEC 13818-6 International Standard ISO/IEC, 1996

[11] Balabanian, V; Casey, L; Greene, N; Adams, C "An Introduction to Digital Storage Media - Command and Control" IEEE Communications Magazine, November 1996, pp. 122-127

[12] The DAVIC 1.4 specification is available via anonymous ftp at ftp://ftp.davic.org/Davic/Pub/Spec1_4/

[13] The TINA-C specifications are available at http://www.tinac.com/ specifications/specifications.htm

[14] Budd, T "An introduction to Object-Oriented Programming" Addison Wesley 1991

[15] Campbell, A; Coulson, G; Hutchison, D "A Quality of Service Architecture" ACM SIGCOMM Computer Communication Review,Vol. 24 No. 2., pp. 6-27, April 1994

[16] Chan, S; Wong, E "Modeling of Video-on-Demand Networks with Server Selection" GlobeCom98 pp. 171-176

[17] Anido, G; Barnett, S "A cost comparison of Distributed and Centralized Approaches to Video-on-Demand" IEEE Journal on Selected Areas in Communications, Vol. 14, No. 6, August 1996 pp. 1173-1183

[18] The Stony Brook Video Server Project at the State University of New York at http://www.ecsl.cs.sunysb.edu/~vernick/sbvs.html

[19] Tsang, R; Du, D; Pavan, A "Experiments with video transmission over an Asynchronous Transfer Mode (ATM) Network" Multimedia Systems (1996) No 4: pp. 157-171

[20] Krunz, M; Tripathi, S "Scene-Based Characterization of VBR MPEG-Compressed Video Traffic," University of Maryland, Department of Computer Science CS-TR-3573, 1996. (Cross-referenced as UMIACS TR-95-120. An abridged version appeared in Sigmetrics '97). Available on the WWW at http://www.ece.arizona.edu/~krunz/tech_rep.html

[21] Rose, O "Statistical properties of MPEG video traffic and their impact on traffic modeling in ATM systems". Proceedings of the 20th Annual Conference on Local Computer Networks, Minneapolis, MN, 1995, pp. 397-406

[22] Srivastava, A; Kumar, A; Singru, A "Design and analysis of a video-on-demand Server" Multimedia Systems 1997 No. 5 pp. 238-254

[23] Coffey, G "Video over ATM Networks" at http://www.cis.ohio-state.edu/~jain/cis788-97/video_over_atm/index.htm

[24] Dubois, D; Georganas, N; Horlait, E "A QoS Selector for Multimedia Applications on ATM Networks" Conference Record: IEEE ICC/SUPERCOMM 1994, pp. 160-164

[25] Murthy, S; Lalgudi, H "Impact of QOS requirements on video coding for ATM networks" Multimedia Systems 1996 No. 4 pp. 316-327

[26] Adler, R "Distributed coordination models for Client/Server computing" IEEE Computer Society, Computer April 1995

[27] Niehaus, D; Battou, A; McFarland, A; Decina, B; Dardy, H; Sirkay, V; Edwards, B "Performance Benchmarking of Signaling in ATM Networks" IEEE Communications Magazine, August 1997. An online version is available at http://hegel.ittc.ukans.edu/docs/ARTS/svc-performance.ps

[28] Aurrecoechea, C; Campbell, A; Hauw, L "A Survey of QoS Architectures" ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture, Vol. 6 No. 3, pp. 138-151, May 1998

[29] Kurose, J "Open Issues and Challenges in Providing Quality of Service Guarantees in High Speed Networks", ACM Computer Communications Review, Vol 23, No 1, pp. 6-15, January 1993

[30] Lazar, A "A Real-time Control, Management and Information Transport Architecture for Broadband Networks", Proceedings International Zurich Seminar on Digital Communications, pp. 281-295, 1992

[31] Volg, C; Wolf, L; Herrtwich, R; Wittig, H "HeiRAT - Quality of Service Management for Distributed Multimedia Systems" Multimedia Systems Journal 1996

[32] TINA-C, "The QoS Framework" Internal Technical Report, 1995

[33] Atiquzzaman, M; Zheng, B "Traffic Management of Multimedia over ATM Networks" IEEE Communications Magazine, January 1999, Vol 37 No 1, pp. 33-38

[34] Chaney, A; Wilson, I; Hopper, A "The Design and Implementation of a RAID-3 Multimedia File Server" Proceedings NOSSDAV95, 5th Workshop on Networking and Operating System Support for Digital Audio and Video, 1995

[35] Vine, P; Ventura, M "Maximizing the simultaneous users of a Video on Demand Server" Proceedings of SATNAC 98, University of Cape Town, 1998

[36] Haskin, R; Schmuck, F. "The Tiger Shark File System" Proceedings of the IEEE 1996 Spring COMPCON, Santa Clara, CA, February 1996

[37] Kwon, T-G; Choi, Y; Lee, S "Disk Placement for arbitrary-rate playback in an interactive video Server" Multimedia Systems (1997) 5: pp. 271-281

[38] Lele, A; Nandy, S "Can QoS guarantees be supported for live video over ATM Networks?" Globecom98 pp. 2315-2323

[39] Murphy, J "Resource Allocation In ATM Networks" available via anonymous ftp at ftp://ftp.eeng.dcu.ie/pub/telecom/murphyj/thesis/thesis.ps

[40] Wu, C; Jiau, J; Chen, K "Characterizing Traffic Behaviour and Providing End-to-End Service Guarantees within ATM Networks" Infocom 97 0-8186-7780-5/97

[41] Windows Sockets 2 Application Programming Interface Revision 2.2.1 May 2,1997

[42] Parekh, A; Gallager, B "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks-the Multiple Node Case," IEEE/ACM Trans. on Networking, Apr. 1994, pp. 137-150

[43] Atiquzzaman, M; Zheng, B "Multimedia over High Speed Networks: Reducing Network Requirements with fast Buffer Fillup" GlobeCom98 pp. 376-381

[44] Vandalore, B; Fahmy, S; Jain, R; Goyal, R; Goyal, M; "QoS and Multipoint Support for Multimedia Applications over the ATM ABR Service" IEEE Communications Magazine, January 1999 pp. 53-57

[45] Feng, W "Video-on-Demand Services: Efficient Transportation and Decompression of Variable Bit Rate Video", PhD Thesis, University of Michigan, April 1996

[46] Dey-Sircar, J; Salehi, J; Kurose, J; Towsley, D "Providing VCR capabilities in large-scale video Servers" Proceedings ACM Multimedia 1994, pg. 25

[47] Ozden, B; Biliris, A; Rastogi, R; Silberschatz, A "A low-cost storage Server for movie on demand databases" Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994

Appendix A: Selected Source Code

Appendix A contains selected source code for the Server, Broker, Client and ToolBox software and focuses on the following classes for each:

In order to reduce the number of pages of code in Appendix A, all Visual Studio-generated code, all code written to manage connections, all code to display dialog boxes for user interaction and all code to play MPEG-1 is available on the accompanying CD but not given here. Furthermore, Visual Studio-generated information has been stripped away from the source files presented in Sections 1,2,3 and 4 as this does not aid in the understanding of the code. The Disk_Manager, DiskReader, FrameReader, Scheduler and Stream_Manager classes of the Server were co-designed and co-authored with Patrick Vine who can be contacted via email at *pvine@microsoft.com*.

In order to use the accompanying CD, place it in a CD drive and click on the file **ATOM.html** which will load in a web browser (Internet Explorer or Netscape Navigator).

1. Server Classes

1.1 ATM.cpp

```
#include "stdafx.h"
#include "ServerVOD.h"
#include "ATM.h"
#include "uuid.h"
#include "Stream_Manager.h"
#include "VideoDatabase.h"
#include "ListConnections.h"

extern VideoDatabase Videos;
char *string;
int bufnum = 0;
extern CStream_Manager SM;
extern ListConnections clients;
BOOL Buffer_Allocation_Array[100];                    // The array used for allocating buffer numbers to new clients

struct atmhost                                        // The structure which holds all information relating to a client

{ char *name;
unsigned char addr[20];
} atmhost_list[20];

CATM::CATM() {}
CATM::~CATM() {}

int CATM::Launch(CString broker_name)
{int fail = 0;
WORD ver_req = MAKEWORD(2, 0);
WSADATA wsa_data;

if (WSAStartup(ver_req, &wsa_data) != 0)
{    AfxMessageBox("Error : Starting up WinSock2.0 DLL");  }

if (LOBYTE(wsa_data.wVersion) != 2 || HIBYTE(wsa_data.wVersion) != 0)
{    AfxMessageBox("Error : Could not find WinSock2.0 DLL");
WSACleanup();  }

SOCKET broker_socket;                                 // Create the broker socket

if ((broker_socket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{  AfxMessageBox("WSASocket Error in creating broker_socket");
WSACleanup();
exit(0); };

DWORD bytes;
bytes = 0;
struct sockaddr_atm sock_addr;
memset((void *)&sock_addr, 0, sizeof(struct sockaddr_atm));
char *name = broker_name.GetBuffer(0);

if (Get_Atm_Address(name, &sock_addr) == FALSE)       // Get the Broker's address from the ATM hosts file
{  AfxMessageBox("Error : ATM NSAP Address not found for server");  }
sock_addr.satm_family          = AF_ATM;
sock_addr.satm_number.AddressType  = ATM_NSAP;
sock_addr.satm_number.NumofDigits  = ATM_ADDR_SIZE;
sock_addr.satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

char convert[2];
UCHAR a;
CString broker_address;
for(int p = 0; p < 20; p ++ )
{  a = sock_addr.satm_number.Addr[p];
```

```
sprintf(convert,"%02x",(char*)a);
broker_address += convert; };
                                                    // Attempt to connect to the broker with the broker_socket
if (WSAConnect(broker_socket, (struct sockaddr FAR *)&sock_addr,
sizeof(struct sockaddr_atm), NULL, NULL, NULL, NULL) == SOCKET_ERROR)
{ AfxMessageBox("Error : Broker Unavailable - please try to connect at a later stage.");
fail = 1; }

if(!fail)
{   WSABUF id;                                      // Send an "S" to inform the broker that it is a server connection
id.len = 1;
id.buf = (char *) malloc(1);
ULONG b = 0 ;
ULONG f = 0;
id.buf = "S";
if((WSASend(broker_socket, &id,1, &b, f, NULL, NULL)) == SOCKET_ERROR)
{ AfxMessageBox("Error sending S character - here's why...");
int y;
char * error;
_itoa(y,error,10);
AfxMessageBox(error);};

struct sockaddr_atm GetMyAddr;                      // Get servers's address in a CString format to send to the server
bytes = 0;
DWORD deviceID = 0;
if(WSAIoctl(broker_socket,SIO_GET_ATM_ADDRESS,(LPVOID)&deviceID,
sizeof(deviceID),(LPVOID)&GetMyAddr,sizeof(GetMyAddr),&bytes,NULL,NULL) == SOCKET_ERROR)
AfxMessageBox("Error getting my ATM address.");

CString server_address;                             // Convert server_address to a CString format
for(p = -4; p < 16; p ++ )
{ a = GetMyAddr.satm_number.Addr[p];
sprintf(convert,"%02x",(char*)a);
server_address += convert; };

CString server_name;
server_name = Get_Name_By_Address(GetMyAddr);       // Go to the atmhosts file, compare this address until we find a
                                                    // match and then we have the machine name of this client
id.len = 20;                                        // Send this server's name to the broker
id.buf = (char *) malloc(20);
b = 0 ;
f = 0;
id.buf = server_name.GetBuffer(20);
if((WSASend(broker_socket, &id,1, &b, f, NULL, NULL)) == SOCKET_ERROR)
{  AfxMessageBox("Error sending client name - here's why...");
int y;
char * error;
_itoa(y,error,10);
AfxMessageBox(error); };

char *sender = server_address.GetBuffer(40);        // Send this server's address to the broker
WSABUF ServerAddress;
ULONG xmitBytes = 0;
ULONG flags = 0;
ServerAddress.len = 40;
ServerAddress.buf = (char *) malloc(40);
ServerAddress.buf = sender;
WSASend(broker_socket, &ServerAddress, 1, &xmitBytes, flags, NULL, NULL);

CString list;                                       // Send the video database to the broker
list=Videos.Create_Video_List_For_Broker();
char *vid_list = list.GetBuffer(10000);
WSABUF VideoList;
VideoList.len = 10000;
VideoList.buf = (char *) malloc(10000);
VideoList.buf = vid_list;
if((WSASend(broker_socket, &VideoList, 1, &xmitBytes, flags, NULL, NULL)) == SOCKET_ERROR)
AfxMessageBox("Error");
```

```
CTime time = CTime::GetCurrentTime();                    // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

Server.Name = server_name;                               // Connected to the broker, fill up the Struct with the details
Server.ATM_Address = server_address;
Server.Time = connection_time;
Server.AAL = "ATMPROTO_AAL5";
Server.Socket_Style = "SOCK_RAW";
Server.Broker_Name = broker_name;
Server.Broker_Address = broker_address;
switch(sock_addr.satm_family)
{case AF_INET:   {Server.Address_Family = "AF_INET";break;} // These address family integers are defined in winsock2.h
case AF_ATM:   {Server.Address_Family = "AF_ATM";break;}
default:    break;  };
switch(sock_addr.satm_number.AddressType)
{case ATM_NSAP:   {Server.Address_Style = "AF_NSAP";break;}
case ATM_E164:   {Server.Address_Style = "AF_E164";break;}
default:    break;  };
char buf[10];
_itoa(broker_socket,buf,10);
Server.Socket_ID = buf;

for(int loop = 0;loop <100;loop++)                       // Start the listening process
Buffer_Allocation_Array[loop] = 0;
AfxBeginThread(ListenToConnectionSocket,NULL,THREAD_PRIORITY_NORMAL);
return 1; }
else return 0; };

void CATM::Close_Winsock()
{   WSACleanup();  }                                     // Close all open sockets and close down Winsock2 gracefully

/*****************************************************************/
/* Function Name:      ListenToConnectionSocket                  */
/* Function:           Listens for client requests to the Spawn Socket,  */
/*                     when a new client wants to join it creates a unique */
/*                     socket and spins the thread               */
/*****************************************************************/

UINT CATM::ListenToConnectionSocket(LPVOID param)
{   SOCKET SpawnSocket;

// Create a spawning socket (that we will listen on)
if ((SpawnSocket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{   AfxMessageBox("WSASocket Error");
WSACleanup();
exit(0); }

struct sockaddr_atm my_addr;                             // Get this server's address
ULONG bytes = 0;
DWORD deviceID = 0;
if(WSAIoctl(SpawnSocket,SIO_GET_ATM_ADDRESS,(LPVOID)&deviceID,
sizeof(deviceID),(LPVOID)&my_addr,sizeof(my_addr),&bytes,NULL,NULL) == SOCKET_ERROR)
AfxMessageBox("Error getting my ATM address.");

struct sockaddr_atm sock_addr;                           // A fix for the 4-byte offset problem.
for(int l = -4; l < 16; l ++ )
{   sock_addr.satm_number.Addr[l+4] = my_addr.satm_number.Addr[l]; };

// Go to the atmhosts file, compare this address until we find a match and then we have the machine name of this client

CString server_name;
server_name = Get_Name_By_Address(sock_addr);
sock_addr.satm_family           = AF_ATM;                // Address Family is ATM
sock_addr.satm_number.AddressType = ATM_NSAP;            // Address Style is network service access point
sock_addr.satm_number.NumofDigits = ATM_ADDR_SIZE;       // ATM_ADDR_SIZE = 20 (defined in ws2atm.h)
sock_addr.satm_blli.Layer2Protocol = SAP_FIELD_ANY;      // Defined in ws2atm.h
sock_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;   // Defined in ws2atm.h
sock_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT; // Defined in ws2atm.h
sock_addr.satm_number.Addr[19] = 0x80;
```

```
// At this point in the code, we have a socket (called "SpawnSocket") which is about to be bound to the address we have placed in
// the sockaddr_atm structure called sock_addr/ Bind the server ATM address to the Socket created by WSASocket()

if (bind(SpawnSocket, (struct sockaddr FAR *)&sock_addr, sizeof(struct sockaddr_atm))== SOCKET_ERROR)
{  AfxMessageBox("Error : bind");
exit(0); }

struct sockaddr_atm ClientAddr;                    // The structure which holds the client's address information
ATM_CONNECTION_ID connID;                          // The structure which holds DeviceNumber,VPI,VCI info
int len;
int sel=0;
SOCKET NewCustomerSocket;

while(1)
{     if(listen(SpawnSocket, 5) == SOCKET_ERROR)
{ int y=WSAGetLastError();
char error[10];
itoa(y,error,10);
AfxMessageBox(error);
WSACleanup();
exit(0);  }

// Accept the client connection request Note: sock_addr is the address of the connecting (or client) machine getsock will be the
// descriptor for the client. Note: we reuse the sock_addr structure, since we have already bound the address that was in it to the
// server so it can be written over

if ((NewCustomerSocket = WSAAccept(SpawnSocket,    // The socket which is listening for connections
(struct sockaddr FAR *)&ClientAddr,                // Pointer to the buffer which receives the address
                                                   // of the connecting entity
&len,                                              // The length of the address
NULL,
0))
== SOCKET_ERROR)
{ AfxMessageBox("Error : accept");  }
bytes = 0;

// After WSAAccept() succeeds and returns a new socket handle, that accepted socket  may not be used to accept more
// connections. The original socket remains open, and listening for new connections
// WSAIoctl: This routine is used to set or retrieve operating parameters associated with the socket

if (WSAIoctl(NewCustomerSocket,                    // The client socket
SIO_GET_ATM_CONNECTION_ID,
NULL,
0,
(LPVOID) &connID,                                  // connID is an ATM_CONNECTION_ID structure which holds
                                                   // DeviceNumber,VPI,VCI info
sizeof(ATM_CONNECTION_ID),
&bytes,
NULL,
NULL)
== SOCKET_ERROR)
{ AfxMessageBox("Error: WSAIoctl");  }

int loop=0;                                        // Create a new bufnum for this new client
while(Buffer_Allocation_Array[loop] != 0)
loop++;
Buffer_Allocation_Array[loop] = 1;
int newone = loop;

Transfer Info;                                     // Pass this client to a ClientComms thread
Info.ThisCustomerSocket = NewCustomerSocket;
Info.bufnum = newone;
Info.DeviceNumber = connID.DeviceNumber;
Info.VCI = connID.VCI;
Info.VPI = connID.VPI;
AfxBeginThread(ClientComms,(LPVOID)&Info,THREAD_PRIORITY_NORMAL);   };
return 0; }
```

```
/*******************************************************/
/* Function Name:        ClientComms                   */
/* Function:             Communicates with a unique client  */
/*******************************************************/

UINT CATM::ClientComms(LPVOID param)
{  Transfer *p;
p =(Transfer*) param;
SOCKET control_socket = p->ThisCustomerSocket;
CATM* Inst = p->PointerToTheInstance;
int thisbufnum = p->bufnum;
SOCKET video_socket;

// It is better to fill in the Client Connection details here than in the listen loop, because it frees up the listen thread.  Receive the
// client's name

CString client_name;
WSABUF ClientName;
ClientName.len = 10;
ClientName.buf = (char *) malloc(10);
ULONG Bytes = 0;
ULONG Flags = 0;
WSARecv(control_socket, &ClientName, 1, &Bytes, &Flags, NULL, NULL);
client_name = ClientName.buf;
client_name.GetBufferSetLength(10);

CString client_address;                                  // Receive the client's address in neat format
WSABUF ClientAddress;
ClientAddress.len = 40;
ClientAddress.buf = (char *) malloc(40);
WSARecv(control_socket, &ClientAddress, 1, &Bytes, &Flags, NULL, NULL);
client_address = ClientAddress.buf;
client_address.GetBufferSetLength(40);

CString temp_copy = client_address;                      // Create an ATM usable address from the CString
char client_addr[20];
int index = 0;
int i=0;
int m=0;
char j;
char k;
while(i!=40)
{ j = GetBinary(temp_copy[i]);
k = GetBinary(temp_copy[i+1]);
j <<= 4;
client_addr[m] = j | k;
i += 2;
m++; };
client_addr[19]=(char)0x90;

CString client;                                          // Code for displaying an ATM address in "neat" format
UCHAR t;
char next_byte[2];
client.Empty();
for( int l = 0; l < 20; l ++ )
{  t = client_addr[l];
sprintf(next_byte,"%02x",(char*)t);
client += next_byte; };

// Create the video socket for video data transmission

if ((video_socket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)

{  AfxMessageBox("WSASocket Error for video_socket");
WSACleanup();
exit(2); };
struct sockaddr_atm sock_addr;
memset((void *)&sock_addr, 0, sizeof(struct sockaddr_atm));
prepare_struct(&sock_addr);
```

```
memcpy(sock_addr.satm_number.Addr,client_addr,ATM_ADDR_SIZE);
Sleep(2000);

// Connect to client

if(WSAConnect(video_socket,(struct sockaddr FAR *) &sock_addr,
sizeof(struct sockaddr_atm),NULL,NULL,NULL,NULL)==SOCKET_ERROR)
{  AfxMessageBox("Error:Connect to Client");
int y=WSAGetLastError();
char * error;
_itoa(y,error,10);
AfxMessageBox(error);
exit(0);  };

char devicenum[10];                                    // Get the DeviceNumber, vci, vpi etc
char vpi[10];
char vci[10];
CString DEVICENUM;
CString VPI;
CString VCI;
CString SOCKETID;
CString BUFNUM;
int q = p->DeviceNumber;
_itoa(q,devicenum,10);
q = p->VCI;
_itoa(q,vci,10);
q = p->VPI;
_itoa(q,vpi,10);
char socket_as_string[10];
_itoa(control_socket,socket_as_string,10);
char client_num_as_string[10];
_itoa(thisbufnum,client_num_as_string,10);

CTime time = CTime::GetCurrentTime();                  // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

DEVICENUM = (CString)devicenum;                        // Add this client to the Client connection list
VPI = (CString)vpi;
VCI = (CString)vci;
SOCKETID = (CString)socket_as_string;
BUFNUM = (CString)client_num_as_string;
CString Video_Selected = "None Selected";
clients.Insert_In_Order(client_name,client_address,connection_time,DEVICENUM,VPI,VCI,SOCKETID,BUFNUM,Video_Selecte
d);

// At this point we have the socket for this client until the client is finished.We need a loop that gets a request from the client and
// deals with it. Possible requests: (1) I want to select a new movie  (2) Play, Stop, Pause, Fast Forward, Rewind, Pause (3) Exit

WSABUF choice;
choice.len = 1;
choice.buf = (char *) malloc(1);
ULONG choiceBytes = 0;
ULONG choiceFlags = 0;
while(1)
{  if (WSARecv(control_socket, &choice, 1, &choiceBytes, &choiceFlags, NULL, NULL) == SOCKET_ERROR)
{
Buffer_Allocation_Array[thisbufnum] = 0;                    // Free up the buffer number
clients.Delete_by_Name(client_name);                        // Remove all details of disconnecting client
closesocket(control_socket);
closesocket(video_socket);
break; }

// What do we send to Stream Manager via DoSM? (1) The control signal (e.g. play, stop, pause, ffwd, rewind, pause)
// (2) The bufnum (3) The socket connected to the client

SM.DoSM(choice.buf, thisbufnum, control_socket, video_socket);  }
return 0; }
char CATM::GetBinary(char the_char)
{   char the_return;
```

```
switch(the_char)
{ case '0':  the_return = 0x0;break;
case '1':  the_return = 0x1;break;
case '2':  the_return = 0x2;break;
case '3':  the_return = 0x3;break;
case '4':  the_return = 0x4;break;
case '5':  the_return = 0x5;break;
case '6':  the_return = 0x6;break;
case '7':  the_return = 0x7;break;
case '8':  the_return = 0x8;break;
case '9':  the_return = 0x9;break;
case 'a':  the_return = 0xa;break;
case 'b':  the_return = 0xb;break;
case 'c':  the_return = 0xc;break;
case 'd':  the_return = 0xd;break;
case 'e':  the_return = 0xe;break;
case 'f':  the_return = 0xf;break;
default: AfxMessageBox("Error in GetBinary()'s case statement");    }
return the_return; };

void CATM::TxSection( int size, BYTE data[], SOCKET video_socket )
{ULONG flags = 0;
WSABUF  xmitBuffer;                          // WSABUF buffer structure for information sent
ULONG recvBytes = 0;
ULONG xmitBytes = 0;
int   pktRecv = 0;
int err = 0;
int pktSize;
counter += size;
pktSize = size;                              // size of each packet sent in bytes


xmitBytes = 0;                               // New connection accepted so prepare to send data
flags = 0;
char sizeSTR[20];                            // A string to hold the size of the next  video block
itoa( size, sizeSTR, 10 );                   // Convert the parameter from Scheduler Module to string to
                                             // send to the client
xmitBuffer.len = 20;                         // Length of each packet (in bytes)
xmitBuffer.buf = sizeSTR;                    // Fill the buffer with the size of the next  video block
if (size > 55000 )                           // Test if video block size exceeds 55 Kbytes
{ char buf2[20];                             // It could cause an overflow error on the internal buffer of the
                                             // FORE adapter
_itoa( size, buf2, 10 );
AfxMessageBox("size is ");
AfxMessageBox(buf2); }

// This first Send command sends the size of the next video block

if (WSASend(video_socket, &xmitBuffer, 1, &xmitBytes, flags, NULL, NULL) == SOCKET_ERROR)
{   AfxMessageBox("Error transmitting size data",MB_OK,0);
WSACleanup();
exit(0);   }

// Now prepare to send the next block of video data. There are two steps:
// (1) Calculate the loop index
// (2) Transmit the video block

char * dataPTR;                              // Create a pointer to a string
dataPTR = (char *) data;                     // Point to the beginning of the next block of video data
int loop = size/240;                         // The loop counter is the size of the video block divided by the
                                             // value 240. This value was found to be optimal by trial and
                                             // error  and estimates about 5 ATM cells
if ((size%240) > 0 )                         // If a remainder occurs in the division then  increment the loop
loop++;                                      // counter by one to cover the extra data
for (int y=0; y<loop; y++)                   // This loop sends the video block in chunks of 240 bytes of data
{   if((y == loop-1) && ( size%240 != 0 ))
xmitBuffer.len = size%240;                   // The length of the transmit buffer is size / 240
else
xmitBuffer.len = 240;
```

```
xmitBuffer.buf = dataPTR;                          // Place the video data into the transmit buffer
dataPTR += xmitBuffer.len;                         // Move the data pointer ahead to the next video block

// This second Send command sends 240 bytes of the next video block

if (WSASend(video_socket, &xmitBuffer, 1, &xmitBytes, flags, NULL, NULL) == SOCKET_ERROR)
{ AfxMessageBox("Error transmitting video data",MB_OK,0);
WSACleanup();
exit(0);   } }
if( xmitBytes != xmitBuffer.len )                  // Check that all data was sent
AfxMessageBox("Wrong length sent!");  }

/*******************************************************/
/* Function Name:      SendFinished                    */
/* Function:           Stop the Client from plaiyng    */
/*******************************************************/

void CATM::SendFinished(SOCKET video_socket)
{ WSABUF Finished;
Finished.len=20;
Finished.buf=(char *) malloc(20);
ULONG xmitBytes = 0;
ULONG Flags = 0;
Finished.buf = "4";
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed");
Finished.len=4;
BYTE data[4] = {0x00, 0x00, 0x01, 0xB9};
char * dataptr = (char*)data;
Finished.buf = dataptr;
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed");
Finished.len=20;
Finished.buf = "Finished";
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed"); }

/*******************************************************/
/* Function Name:      Receive_Movie_Choice            */
/* Function:           To receive the client's choice of movie */
/*******************************************************/

char * CATM::Receive_Movie_Choice(SOCKET control_socket)

{  WSABUF choice;
char * error_message="Receive Error";
choice.len = 50;
choice.buf = (char *) malloc(50);
ULONG choiceBytes = 0;
ULONG choiceFlags = 0;

if (WSARecv(control_socket, &choice, 1, &choiceBytes, &choiceFlags, NULL, NULL) == SOCKET_ERROR)
{ AfxMessageBox("Error receiving choice in Receive Movie Choice");
return error_message;  }
return choice.buf; }

BYTE CATM::NewBufMax(BYTE BufNum)
{   while(( Buffer_Allocation_Array[BufNum] == 0 ) && ( BufNum > 0 ))
BufNum--;
return BufNum; }

BOOL CATM::prepare_struct(struct sockaddr_atm *the_struct)
{ the_struct->satm_family           = AF_ATM;
the_struct->satm_number.AddressType  = ATM_NSAP;
the_struct->satm_number.NumofDigits  = ATM_ADDR_SIZE;
the_struct->satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
the_struct->satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
the_struct->satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;
```

```
return TRUE; }

CString CATM::Get_Name_By_Address(struct sockaddr_atm address)
{ char* the_name;
CString name;
CString error = "Error";
int i;
int atmhost_nb = 0;
char *filename = "D:\\winnt\\atmhosts";
Read_File(filename, atmhost_nb);
for (i=0;i<atmhost_nb;i++)
{   atmhost_list[i].addr[19] = (char) 0x0;
if (memcmp(address.satm_number.Addr - 4,atmhost_list[i].addr,20)==0 && strcmp("localhost",atmhost_list[i].name)!=0)
{
the_name = atmhost_list[i].name;
name = CString(the_name);
return name;      };      };
return error; }

void CATM::Read_File(const char* name, int & atmhost_nb)
{ FILE *fp;
char buffer[1024];
fp = fopen(name,"r");
if (fp == NULL)
{  AfxMessageBox("File Open Error in Read_File");
return;   }

while (fgets(buffer,sizeof(buffer),fp)!= NULL)
{  if (buffer[0] == ';' || buffer[0]=='#')
continue;
Parse_Line(buffer,atmhost_nb);   }
fclose(fp); }

void CATM::Parse_Line(char *buffer, int & atmhost_nb)
{ int i,j,n,k;
unsigned char addr[20];
char *start, *end;

// i: index into the parsed line  j: inder into addr  k: number of figure currently decoded ;  remove trailing \r and \n

start = strrchr(buffer,'\r');
if (start != NULL) *start = 0;
start = strrchr(buffer,'\n');
if (start != NULL) *start = 0;

if (*buffer == 0) return;                                   // Skip empty lines
memset(&addr,0,sizeof(addr));
n = strlen(buffer);
j = k = 0;
for (i=0;i<n;i++)
{   if ((buffer[i]>='0'&&buffer[i]<='9')
|| (buffer[i]>='a'&&buffer[i]<='f')
|| (buffer[i]>='A'&&buffer[i]<='F'))
{    if (k>=2)                                              // We already read two figures, so read the next atm addr byte
{  j++;
k = 0;   }

if (j>=20)
{ AfxMessageBox("address too long");
return;  }
addr[j] = addr[j]*16 + Get_Hex(buffer[i]);
k++;   }

else if (buffer[i]=='.'||buffer[i]=='-')
{ k = 0;
j++;
if (j>=20)
{
AfxMessageBox("invalid address");
```

```
return;  }  }
else  {
if (j==20 || (j==19 && k>0))
break;
AfxMessageBox("address too short");
return;  }  }

start = buffer+i;                                    // addr is now correctly filled in, need to get the name
while ((*start==' '||*start=='\t') && *start!=0)
start++;
if (*start == 0)
{ AfxMessageBox("no name");
return;  }
end = start;
while (*end!=' '&&*end!='\t'&&*end!='\r'&&*end!='\n'&&*end!=0)
end++;
*end = 0;
atmhost_list[atmhost_nb].name = strdup(start);
memcpy(atmhost_list[atmhost_nb].addr,addr,20);
atmhost_nb ++; }

int CATM::Get_Hex(char c)
{  if (c>='0' && c<='9')
return (c-'0');
if (c>='a' && c<='f')
return (c-'a'+10);
if (c>='A' && c<='F')
return (c-'A'+10);
printf("internal error in name.c\n");
return 0; }

/*****************************************************************/
/* Function Name:      get_atm_address                           */
/* Function:           Resolve the name string such as "server" into */
/*                     an actual ATM address                     */
/*****************************************************************/

BOOL CATM::Get_Atm_Address(char *name, struct sockaddr_atm *atm_addr)
{DWORD dwValue;
HANDLE hLookup;
WSAQUERYSETW qsRestrictions;
CSADDR_INFO  csaBuffer;
WCHAR  tmpWStr[100];
MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, name, -1, tmpWStr, 100);
csaBuffer.LocalAddr.iSockaddrLength = sizeof (struct sockaddr_atm);
csaBuffer.LocalAddr.lpSockaddr = (struct sockaddr *) atm_addr;
csaBuffer.RemoteAddr.iSockaddrLength = sizeof (struct sockaddr_atm);
csaBuffer.RemoteAddr.lpSockaddr = (struct sockaddr *) atm_addr;
qsRestrictions.dwSize               = sizeof (WSAQUERYSETW);
qsRestrictions.lpszServiceInstanceName = NULL;
qsRestrictions.lpServiceClassId      = &FORE_NAME_CLASS;
qsRestrictions.lpVersion             = NULL;
qsRestrictions.lpszComment           = NULL;
qsRestrictions.dwNameSpace           = FORE_NAME_SPACE;
qsRestrictions.lpNSProviderId        = NULL;
qsRestrictions.lpszContext           = L"";
qsRestrictions.dwNumberOfProtocols   = 0;
qsRestrictions.lpafpProtocols        = NULL;
qsRestrictions.lpszQueryString       = tmpWStr;
qsRestrictions.dwNumberOfCsAddrs     = 1;
qsRestrictions.lpcsaBuffer           = &csaBuffer;
qsRestrictions.lpBlob                = NULL;

if (WSALookupServiceBeginW(&qsRestrictions, LUP_RETURN_ALL, &hLookup) == SOCKET_ERROR)
{ AfxMessageBox("Socket error : WSALookupServiceBegin");
return FALSE;  }
dwValue = sizeof (WSAQUERYSETW);

if (WSALookupServiceNextW ( hLookup, 0, &dwValue, &qsRestrictions) == SOCKET_ERROR)
```

```
{ AfxMessageBox("Socket error : WSALookupServiceNext");
return FALSE; }

if (WSALookupServiceEnd (hLookup) == SOCKET_ERROR)
{ AfxMessageBox("Socket error : WSALookupServiceEnd");
return FALSE; }
return TRUE; }
```

1.2 ATM.h

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <ws2atm.h>
#include <ws2spi.h>
#define FORE_NAME_SPACE NS_ALL

class CATM : public CFrameWnd
{ public:
CATM();
void TxSection( int size, BYTE data[],SOCKET video_socket );
char * Receive_Movie_Choice(SOCKET control_socket);
void SendFinished(SOCKET video_socket);
static char GetBinary(char the_char);
static CString Get_Name_By_Address(struct sockaddr_atm address);
static void Parse_Line(char *buffer, int & atmhost_nb);
static void Read_File(const char* name,int & atmhost_nb);
static int Get_Hex(char c);

virtual ~CATM();

struct ServerDetails
{ CString Name;
CString ATM_Address;
CString Socket_ID;
CString Address_Family;
CString Address_Style;
CString AAL;
CString  Socket_Style;
CString Time;
CString Broker_Name;
CString Broker_Address;
} Server;

struct Listener
{  CATM* PointerToTheInstance;
SOCKET SpawningSocket;
};

struct Transfer

{  CATM* PointerToTheInstance;
SOCKET ThisCustomerSocket;
int bufnum;
DWORD DeviceNumber;
DWORD VPI;
DWORD VCI;
};

public:
BYTE NewBufMax( BYTE BufNum );
int counter;
void Close_Winsock();
int Launch(CString broker_name);
static UINT ListenToConnectionSocket(LPVOID param);
static UINT ClientComms(LPVOID param);
BOOL Get_Atm_Address(char *name, struct sockaddr_atm *atm_addr);
```

```
private:
static BOOL get_atm_address( char *, struct sockaddr_atm * );
static BOOL prepare_struct(struct sockaddr_atm *); };
```

1.3 Disk_Manager.cpp

```
#include "stdafx.h"
#include "Disk_Manager.h"
#define MAXCLIENTS 20
#include "EvalTools.h"
#include <mmsystem.h>
CEvaluate DMEval;                                   // Saves time to read from disk
#define MAXCLIENTS 20
const unsigned int TXSIZE = 7623;
#define SMOOTHING_TYPE 1

// 1 = 1 frame at a time  2 = 2 frames ... -1 = I-I macroframing -2 = I-P / P-P / P-I macroframing

CDisk_Manager::CDisk_Manager( CScheduler * Scheduler )   // Constructor initialises DMEvent to unsignaled
{  ListHead = NULL;
ListTail = NULL;
DMEvent.ResetEvent();
DMEval.OpenFile("DM_read_times.txt"); }

CDisk_Manager::~CDisk_Manager()                     // Deletes all memory allocated and closes all files open

{   LIST * Temp;
CloseAllFiles();
while( ListHead != NULL )                           // Run through fill-buffer list ensuring all memory  deallocated

{Temp = ListHead;
ListHead = ListHead-NextNode;
delete Temp;   } }

// Adds a new buffer to the buffer list and releases the ReadDM thread to process it
void CDisk_Manager::AddABufferToList(BUFFERINFO * Buf, int BufNum)
{ LIST * NewNode = new LIST;                        // Create a new node
NewNode-NextNode = NULL;                            // Allocate the variables in the node
NewNode-BNum = BufNum;
NewNode-Buffer = Buf;
if (ListHead == NULL)                               // If the list is empty, create one node list
{ ListHead = NewNode;
ListTail = NewNode;  }
else
{ListTail-NextNode = NewNode;                       // Otherwise add node to end of list
ListTail = NewNode;   }
DMEvent.SetEvent();  }                              // Signal the ReadDM thread to run


// Add a new client to the disk manager
void CDisk_Manager::AddToDM(int BufNum, char * MovieName, BUFFERINFO * Buf )
{ MMRESULT mmres = timeBeginPeriod( 1 );
int size;
int numframes = SMOOTHING_TYPE;
Buf-Crit_Section.Lock();                            // Lock the buffers critical section.
Frame[BufNum].OpenMPEGFile( MovieName );            // Initialise the macroframe reading object - open files, etc.
DWORD startTime = timeGetTime();                    // Gets start time

// This code used for macroframe type smoothing. SMOOTHING_TYPE defined at top of file. -1 = macroframing, -2 = IP
// macroframing, 1 = single frame at a time, n = n frames read in at a time - transmit rate = amount of data/n

numframes = SMOOTHING_TYPE;
// Read a macroframe, size = size of the macroframe and the number of frames in the macroframes appears in numframes
size = Frame[BufNum].GetMacroFrame( Buf-Start, &numframes, &Buf-LastBuffer );

// This code used for blocking type transmission. only constant bit rate has been defined as yet. TXSIZE defined at top of file
// numframes = TXSIZE * 8;
// size = Frame[BufNum].GetBlock( numframes, Buf-Start, &Buf-LastBuffer );
```

```
DWORD endTime = timeGetTime();                    // Gets end time
DMEval.SetData( endTime - startTime );            // Stores data to be saved - times time to get macroframe/block

if (size  0 )                                     // If some data was read
{
// Set up the buffer as it should be
Buf-End = Buf-Start + size;                       // End of data in buffer

// This code used for macroframe type smoothing

Buf-PacketSize = size/numframes;                  // New packet size to schedule
if (size%numframes != 0 )                         // Round packet size up
Buf-PacketSize++;

// This code used for blocking type transmission
// Buf-PacketSize = TXSIZE;

Buf-Current_Pos = Buf-Start;                      // Set current position
Buf-Accessible = TRUE;                            // Make the buffer accessible
}
Buf-Crit_Section.Unlock();                        // Unlock the crit. section
timeEndPeriod( 1 );
}

// The ReadDM thread - the main method
void CDisk_Manager::ReadDM()
{ MMRESULT mmres = timeBeginPeriod( 1 );
int BufNum, size, numframes;
numframes = SMOOTHING_TYPE;
LIST * Temp;
Quit = FALSE;
while(!Quit)                                      // Loop continuously until told to quit with the Quit variable

{
// Waits for the DMEvent to be signaled by the AddABufferToList method
::WaitForSingleObject(DMEvent.m_hObject, INFINITE );
DMEvent.ResetEvent();                             // Reset the DMEvent object
while( ListHead != NULL )                         // While the list is not empty
{
BufNum = ListHead-BNum;                           // Get the client's ID

// Read a macroframe, size = size of the macroframe and the number of frames in the macroframes appears in numframes
DWORD startTime = timeGetTime();                  // Get start time

// This code used for macroframe type smoothing. SMOOTHING_TYPE defined at top of file.  -1 = macroframing, -2 = IP
// macroframing, 1 = single frame at a time, n = n frames read in at a time - transmit rate = amount of data/n

numframes = SMOOTHING_TYPE;
size = Frame[BufNum].GetMacroFrame( ListHead-Buffer-Start, &numframes, &ListHead-Buffer-LastBuffer );


// This code used for blocking type transmission. only constant bit rate has been defined as yet. TXSIZE defined at top of file
//      numframes = TXSIZE * 8;
//      size = Frame[BufNum].GetBlock( numframes, ListHead-Buffer-Start, &ListHead-Buffer-LastBuffer );

DWORD endTime = timeGetTime();                    // Get start time
DMEval.SetData( endTime - startTime );            // Stores data to be saved - times time to get macroframe/block

if( size  0 )                                     // If some data was read
{  // Set up the buffer as it should be
ListHead-Buffer-Crit_Section.Lock();              // Lock the crit. section
ListHead-Buffer-End = ListHead-Buffer-Start + size;  // End of data in buffer

// This code used for macroframe type smoothing

if( numframes == 0 )
numframes++;
ListHead-Buffer-PacketSize = size/numframes;      // New packet size to schedule
```

```
if (size%numframes != 0 )                          // Round packet size up
ListHead-Buffer-PacketSize++;

// This code used for blocking type transmission
//        ListHead-Buffer-PacketSize = TXSIZE;

ListHead-Buffer-Current_Pos = ListHead-Buffer-Start;   // Set current position
ListHead-Buffer-Accessible = TRUE;                     // Make the buffer accessible
ListHead-Buffer-Crit_Section.Unlock();                 // Unlock the crit. section
}
Temp = ListHead;                                   // Then remove the buffer from the list and delete it
ListHead = ListHead-NextNode;
delete Temp;      }   }
timeEndPeriod( 1 ); }

void CDisk_Manager::CloseAllFiles()                // Method to close all files that were opened in the Frame array
{   for( int loop=0; loop<MAXCLIENTS; loop++ )
Frame[loop].CloseMPEGFile(); }

void CDisk_Manager::SetQuit()                      // Method to end ReadDM thread
{  Quit = TRUE; }

void CDisk_Manager::CloseStream(int BufNum)        // Closes a stream that was opened
{   Frame[ BufNum ].CloseMPEGFile(); }
```

1.4 Disk_Manager.h

```
#include "Globals.h"                               // Struct definitions
#include "afxmt.h"                                 // For CEvent class and the threads
#include "FrameReader.h"                           // Added by ClassView

class CDisk_Manager
{
public:
void CloseStream( int BufNum );                    // Closes an opened stream
void SetQuit();                                    // Sets quit to allow the ReadDM thread to end
void CloseAllFiles();                              // Closes all files open in the Frame array
void ReadDM();                                     // main thread for refilling buffers
void AddToDM( int BufNum, char * MovieName, BUFFERINFO * Buf );      // Add a new client to the disk manager
void AddABufferToList( BUFFERINFO * Buf, int BufNum );   // Add a buffer to be filled to the buffer list
CDisk_Manager( CScheduler * Scheduler );
virtual ~CDisk_Manager();
private:
CFrameReader Frame[20];
CEvent DMEvent;
BOOL Quit;
LIST * ListTail;
LIST * ListHead; };
```

1.5 DiskReader.cpp

```
// How this file works: It aims to have a disk read-ahead mechanism to speed things up. Whenever the DoRead function is called
// from a FrameReader object, the ReadEvent is signalled and the ReadAhead function (which is in a loop) gets the event and
// reads ahead

#include "stdafx.h"
#include "Toolbox.h"
#include "DiskReader.h"
#include "FrameReader.h"
#define MPEG_BUFFERSIZE 131072                     //The size of the read ahead buffer.

DiskReader::DiskReader() {}

DiskReader::DiskReader(BYTE * MPEGFileBuffer, FrameReader * Frame)
{ DiskBuffer = MPEGFileBuffer;
FrmReader = Frame;
ReadCounter = 0;
```

```
ReadEvent.ResetEvent();
InBuffer = 0;
FileFinished = FALSE; }

DiskReader::~DiskReader()
{ if( m_hFile != INVALID_HANDLE_VALUE )
CloseFile(); }

void DiskReader::DoRead()
{ if( !FileFinished )
{
ReadCounter++;
if( ReadCounter  10 )
int i = 0;
ReadEvent.SetEvent(); }    }

void DiskReader::ReadAhead()
{ while(1)
{
::WaitForSingleObject( ReadEvent.m_hObject, INFINITE );
ReadEvent.ResetEvent();

while( ReadCounter  0 )
{
ReadCounter--;
ULONG AmtRead;

if( m_hFile != INVALID_HANDLE_VALUE )
{
ReadFile( m_hFile; DiskBuffer+InBuffer*(MPEG_BUFFERSIZE/2),
MPEG_BUFFERSIZE/2, &AmtRead, NULL );
if( AmtRead < MPEG_BUFFERSIZE/2 )
{ FileFinished = TRUE;
FrmReader-SetFileFinished();
FrmReader-SetEndOfMPEGBuffer( AmtRead+InBuffer*(MPEG_BUFFERSIZE/2)+2 );
ReadCounter = 0; }
else
FrmReader-SetEndOfMPEGBuffer( AmtRead+InBuffer*(MPEG_BUFFERSIZE/2) );
InBuffer++;
if( InBuffer1 )
InBuffer = 0;        }      }    } }

void DiskReader::FillTotalBuffer()
{ FileFinished = FALSE;
ULONG AmtRead;
SetFilePointer( m_hFile, 0, 0, FILE_BEGIN );
InBuffer = 0;
if( m_hFile != INVALID_HANDLE_VALUE )
ReadFile( m_hFile, DiskBuffer, MPEG_BUFFERSIZE, &AmtRead, NULL ); }

void DiskReader::OpenFile( char * FileName )
{ m_hFile = CreateFile(    FileName,
GENERIC_READ,
FILE_SHARE_READ,
NULL,
OPEN_EXISTING,
0,
NULL);
if (m_hFile == INVALID_HANDLE_VALUE)
AfxMessageBox("CDiskReader::OpenFile():- Could not open MPEG file!"); }

void DiskReader::CloseFile()
{ CloseHandle( m_hFile );
m_hFile = INVALID_HANDLE_VALUE; }
```

1.6 DiskReader.h

```
#include "afxmt.h"                                    // For CEvent class and threads
class FrameReader;
```

```
class DiskReader : public CFrameWnd
{
public:
DiskReader();
DiskReader(BYTE * MPEGFileBuffer, FrameReader * Frame);
~DiskReader();
void FillTotalBuffer();
void ReadAhead();
void DoRead();
void CloseFile();
void OpenFile( char * FileName );
private:
BOOL End;
BOOL FileFinished;
BYTE * DiskBuffer;
FrameReader   * FrmReader;
volatile BYTE ReadCounter;
CEvent ReadEvent;
HANDLE m_hFile;
BYTE InBuffer; };
```

1.7 FrameReader.cpp

```
#include "FrameReader.h"
#include "DiskReader.h"
#define MPEG_BUFFERSIZE 131072                    //The size of the read ahead buffer
#define P_MacroFrame -2
#define I_MacroFrame -1
volatile int count;

CFrameReader::CFrameReader()                      // Constructor - intialise the variables
{ MPEGFileBufferEnd = MPEG_BUFFERSIZE;            // Setup max buffer size
// Below are two checks to ensure if frame reading occurs before file opened - it will return nothing
FileFinished = TRUE;
MPEGFileBufferPosition = MPEGFileBufferEnd;
FillNextBuffer = FALSE;
counter = 0;
Locked = FALSE;
count = 0; }

CFrameReader::~CFrameReader()                     // Destructer - remove memory stuff
{  if( MPEGFileBuffer != NULL )
delete [] MPEGFileBuffer; }

// Manipulates the bit level file to get a frame. In all cases except the first time entering the procedure, the LookAheadBuffer
// will be on a 0x00 00 01 border, thus this must be accounted for

UINT CFrameReader::GetFrame( BYTE *pFrame )
{  BYTE * FramePTR = pFrame;
UINT FrameSize = 0;

if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{     return 0;    }

while(!(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd )))
{   if(First)
First = FALSE;
else
FramePTR = MoveOffCurrentStartCode( FramePTR, &FrameSize );
FramePTR = FindNextStartCode( FramePTR, &FrameSize, 999999 );

if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{ if( LookAheadBuffer[3] != 0xB9 )
{
*FramePTR++ = 0x00;
*FramePTR++ = 0x00;
```

```
*FramePTR++ = 0x01;
*FramePTR++ = 0xB9;
FrameSize += 4;
return FrameSize; }      }

if( LookAheadBuffer[3] == 0x00 )                    // Frame beginning
return FrameSize;   }
return FrameSize;   }

// Manipulate the bit level file to get a macroframe
UINT CFrameReader::GetMacroFrame(BYTE * pMacroFrame, int * amount, BOOL * LastBuffer )
{ UINT VideoSize = 1;
UINT VSumSize = 0;
int NumFrames = 0;
if( *amount == 0 )
return 0;

if( *amount == I_MacroFrame )  // -1
{do
{VideoSize = GetFrame( (pMacroFrame+VSumSize));
VSumSize += VideoSize;
if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
NumFrames++;
if(( First < 2 ) && ((LookAheadBuffer[5] & 0x18) == 0x08 ))
{ First++;
VideoSize = GetFrame( (pMacroFrame+VSumSize));
VSumSize += VideoSize;
if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
NumFrames++;        }     }
while(( (LookAheadBuffer[5] & 0x18) != 0x08 ) &&
( VideoSize  0 ) &&
(*(pMacroFrame+VSumSize-1) != 0xB9 ));
}
else
if( *amount == P_MacroFrame )
{ do
{
VideoSize = GetFrame( (pMacroFrame+VSumSize));
VSumSize += VideoSize;
if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
NumFrames++;
if(( First < 2 ) && ((LookAheadBuffer[5] & 0x18) == 0x08 ))
{  First++;
VideoSize = GetFrame( (pMacroFrame+VSumSize));
VSumSize += VideoSize;
if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
NumFrames++;   }
}
while( ( (LookAheadBuffer[5] & 0x18) != 0x08) &&
( (LookAheadBuffer[5] & 0x18) != 0x10) &&
( VideoSize  0 ) &&
(*(pMacroFrame+VSumSize-1) != 0xB9 ));
}
else
{
do
{
VideoSize = GetFrame( (pMacroFrame+VSumSize));
counter += VideoSize;
VSumSize += VideoSize;
if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
NumFrames++;
}
while (((*amount)  NumFrames ) &&
```

```
(VideoSize  0 ) &&
(*(pMacroFrame+VSumSize-1) != 0xB9 )); }

if(( NumFrames == 0 ) && ( VSumSize  0 ) &&
( FileFinished ) &&
( MPEGFileBufferPosition == MPEGFileBufferEnd ))
*amount = 1;
else
*amount = NumFrames;

if((( *amount == 1 ) || ( VideoSize == 0 )) && ( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{  counter = 0;
*LastBuffer = TRUE;
ResetMPEGFile();
}return VSumSize; }

UINT CFrameReader::StartDisk( LPVOID pParam )
{
CDiskReader * TheDisk;
TheDisk = (CDiskReader *) pParam;
TheDisk-ReadAhead();
return 0; }

void CFrameReader::OpenMPEGFile( char * MovieName )        // Opens the MPEG file and creates a handle to it
{
MPEGFileBufferPosition = 0;                                // Allocate the buffer
MPEGFileBufferEnd = MPEG_BUFFERSIZE;
First = TRUE;
MPEGFileBuffer = new BYTE[MPEG_BUFFERSIZE];
if( MPEGFileBuffer == NULL )
AfxMessageBox("CFrameReader::OpenMPEGFile() - Not enough memory to allocate 2K for MPEGFileBuffer" );
else
if( !(VirtualLock( MPEGFileBuffer, MPEG_BUFFERSIZE ) ))
AfxMessageBox("Locking of MPEGFileBuffer not successful!");
disk = new CDiskReader( MPEGFileBuffer, this );
AfxBeginThread( StartDisk, disk, THREAD_PRIORITY_TIME_CRITICAL );
FileFinished = FALSE;
disk-OpenFile( MovieName );                                // Must fill the buffer initially
disk-FillTotalBuffer();

// Load Initial values into Look Ahead Buffer
LookAheadBuffer[0] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[1] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[2] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[3] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[4] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++]; }

void CFrameReader::CloseMPEGFile()                         // Closes the MPEG file
{disk-CloseFile();
delete disk;
FileFinished = TRUE;
if( ! VirtualUnlock( MPEGFileBuffer, MPEG_BUFFERSIZE ) )
AfxMessageBox( " Unlocking of MPEGFileBuffer not successful!");
delete [] MPEGFileBuffer;
MPEGFileBuffer = NULL;
MPEGFileBufferPosition = MPEGFileBufferEnd;  }             // if call frame read methods, will return nothing


void CFrameReader::ResetMPEGFile()
{ First = TRUE;
disk-FillTotalBuffer();
FileFinished = FALSE;
MPEGFileBufferEnd = MPEG_BUFFERSIZE;
MPEGFileBufferPosition = 0;
LookAheadBuffer[0] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[1] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[2] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[3] = MPEGFileBuffer[MPEGFileBufferPosition++];
```

```
LookAheadBuffer[4] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++]; }

UINT CFrameReader::GetBlock(UINT size, BYTE * pBlock, BOOL * LastBuffer )
{ UINT count = 0;
BYTE * FramePTR = pBlock;
if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
ResetMPEGFile();                                    // Set File back to beginning to fill next buffer

while(( count < size ) && (!(( FileFinished ) &&
( MPEGFileBufferPosition == MPEGFileBufferEnd ))))
{
if(!(( FileFinished ) &&
( MPEGFileBufferPosition == MPEGFileBufferEnd )))
FramePTR = MoveOffCurrentStartCode( FramePTR, &count );

FramePTR = FindNextStartCode( FramePTR, &count, size - count );

if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
if( LookAheadBuffer[3] != 0xB9 )
{
*FramePTR++ = 0x00;
*FramePTR++ = 0x00;
*FramePTR++ = 0x01;
*FramePTR++ = 0xB9;
count += 4;
*LastBuffer = TRUE;                                 // Only occurs when it is last buffer
return count; }}

if(( count < size ) || (( count == size ) && ( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd )))
{*LastBuffer = TRUE;
ResetMPEGFile();}
return count; }

BYTE * CFrameReader::SkipStuffBytes( BYTE * FramePTR, UINT * FrameSize )
{for( int loop = 0; loop <= 5; loop++ )
{LookAheadBuffer[loop] = MPEGFileBuffer[MPEGFileBufferPosition++];
if( MPEGFileBufferPosition == MPEGFileBufferEnd )    // Test Empty condition
{
if( Locked )
{ConflictSection.Unlock();
Locked = FALSE;
}
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0); }

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0; }

while((LookAheadBuffer[0] == 0xFF) || (LookAheadBuffer[0] == 0x0F))
{
LookAheadBuffer[0] = LookAheadBuffer[1];            // Skip stuff bytes
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{if( FileFinished )
{*FramePTR++ = LookAheadBuffer[0];
*FramePTR++ = LookAheadBuffer[1];
*FramePTR++ = LookAheadBuffer[2];
```

```
*FramePTR++ = LookAheadBuffer[3];
*FrameSize += 4;
return FramePTR; } }

if( Locked )
{ConflictSection.Unlock();
Locked = FALSE; }

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{ if( Locked )
{ ConflictSection.Unlock();
Locked = FALSE; }
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0); }

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0; }
return FramePTR; }

BYTE * CFrameReader::MoveOffCurrentStartCode(BYTE * FramePTR, UINT * FrameSize )
{*FramePTR++ = LookAheadBuffer[0];
*FrameSize += 1;

LookAheadBuffer[0] = LookAheadBuffer[1];
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
if( FileFinished )
{ *FramePTR++ = LookAheadBuffer[0];
*FramePTR++ = LookAheadBuffer[1];
*FramePTR++ = LookAheadBuffer[2];
*FramePTR++ = LookAheadBuffer[3];
*FrameSize += 4;
return FramePTR; }

if( Locked )
{ConflictSection.Unlock();
Locked = FALSE; }

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{if( Locked )
{ConflictSection.Unlock();
Locked = FALSE; }
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0); }

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0;

return FramePTR; }
```

// MaxFetch used in conjunction with GetBlock to limit amount fetched to amount wanted - in other calls the function use
// 999999 to ensure SCode found before 999999 bytes 999999 = larger than 640*480*3 byte uncompressed image

```
BYTE * CFrameReader::FindNextStartCode(BYTE * FramePTR, UINT * FrameSize, UINT MaxFetch )
{while(!((LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 )))
{if( MaxFetch == 0 )
return FramePTR;

*FramePTR++ = LookAheadBuffer[0];
*FrameSize += 1;
MaxFetch--;

LookAheadBuffer[0] = LookAheadBuffer[1];
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
if( FileFinished )
{*FramePTR++ = LookAheadBuffer[0];
*FramePTR++ = LookAheadBuffer[1];
*FramePTR++ = LookAheadBuffer[2];
*FramePTR++ = LookAheadBuffer[3];
*FrameSize += 4;
return FramePTR;}

if( Locked )
{ConflictSection.Unlock();
Locked = FALSE; }

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{if( Locked )
{ConflictSection.Unlock();
Locked = FALSE; }
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0); }

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0;
} return FramePTR; }

void CFrameReader::SetEndOfMPEGBuffer( DWORD Position )
{ ConflictSection.Lock();
MPEGFileBufferEnd = Position;
ConflictSection.Unlock(); }

void CFrameReader::SetFileFinished()
{ FileFinished = TRUE; }

UINT CFrameReader::GetAudioFrame(BYTE * pAudioFrame, int * NumFrames, BOOL * LastBuffer )
{ BYTE * FramePTR = pAudioFrame;
UINT FrameSize = 0;
*NumFrames = 0;

if(( FileFinished ) && ( MPEGFileBufferPosition = MPEGFileBufferEnd ))
{ResetMPEGFile();                                    // Set File back to beginning to fill next buffer
}

while(!(( FileFinished ) && ( MPEGFileBufferPosition = MPEGFileBufferEnd )))
{if(First)
First = FALSE;
else
```

```
FramePTR = MoveOffCurrentStartCode( FramePTR, &FrameSize );
FramePTR = FindNextStartCode( FramePTR, &FrameSize, 999999 );

if( LookAheadBuffer[3] == 0xC0 )
return FrameSize;

if( LookAheadBuffer[3] == 0xBE )                    // Stuff Packet that can be skipped
FramePTR = SkipStuffBytes( FramePTR, &FrameSize );

if( LookAheadBuffer[3] == 0x00 )                    // Frame beginning
if((LookAheadBuffer[5] & 0x18) != 0x00)
*NumFrames += 1;

if(( FileFinished ) && ( MPEGFileBufferPosition = MPEGFileBufferEnd ))
if( LookAheadBuffer[3] != 0xB9 )
{ *FramePTR++ = 0x00;
*FramePTR++ = 0x00;
*FramePTR++ = 0x01;
*FramePTR++ = 0xB9;
FrameSize += 4; } }

if(( FileFinished ) && ( MPEGFileBufferPosition = MPEGFileBufferEnd ))
*LastBuffer = TRUE;
return FrameSize; }
```

1.8 FrameReader.h

```
#include "afxmt.h"
#include "stdafx.h"
class CDiskReader;
class CScheduler;

struct TxAudio
{BYTE FrameNumber;
UINT AudioSize;
TxAudio * NextPacket;};

class CframeReader                                  // Class to read frames and macroframes into a buffer using
                                                    // looking at bit level
{public:
static UINT StartDisk( LPVOID pParam );
void SetEndOfMPEGBuffer( DWORD Position );
void SetFileFinished();
void ResetMPEGFile();
UINT GetAudioFrame( BYTE * pAudioFrame, int * NumFrames, BOOL * LastBuffer );
UINT GetBlock( UINT size, BYTE * pBlock, BOOL * LastBuffer );
CFrameReader();
~CFrameReader();
UINT GetFrame( BYTE * pFrame );                     // Gets a frame from the file/buffer
UINT GetMacroFrame( BYTE * pMacroFrame, int * amount, BOOL * LastBuffer );  // Gets a macroframe from the file/buffer
void OpenMPEGFile( char * MovieName );              // Opens the MPEG file and creates a handle to it
void CloseMPEGFile();                               // Closes the MPEG file
private:
CDiskReader * disk;
int counter;
BOOL FillNextBuffer;                                // Used for Macroframing - Defines to return 0 and show that
                                                    // file has been finished, and then to fill the next buffer avail. to
                                                    // be ready next time there is a request for a play by the client
BYTE BufNum;
BYTE * FindNextStartCode( BYTE * FramePTR, UINT * FrameSize, UINT MaxFetch );
BYTE * MoveOffCurrentStartCode( BYTE * FramePTR, UINT * FrameSize );
BYTE * SkipStuffBytes( BYTE * FramePTR, UINT * FrameSize );
BOOL FileFinished;
BOOL First;
BYTE LookAheadBuffer[6];                            // Frame header = 4 bytes then 1 byte then frame type byte
UINT MPEGFileBufferEnd;
HANDLE hMPEGFile;                                   // Handle to the MPEGFile being played
BYTE * MPEGFileBuffer;                              // Buffer for data from the MPEG file
```

```
UINT MPEGFileBufferPosition;                    // Position in the buffer
CScheduler * Sched;
CCriticalSection ConflictSection;
BOOL Locked; };
```

1.9 Scheduler.cpp

```
#include "stdafx.h"
#include "ServerVOD.h"
#include "Scheduler.h"
#include "ATM.h"
#include "EvalTools.h"
#include "Disk_Manager.h"
#include <mmsystem.h>
#include <math.h>
#define MAXCLIENTS 20

                                                // 33.37 for 30 frame per second videos
                                                // 41.64 for 24 frame per second videos
const double FRAMERATE = 33.37;
 extern CATM ATM;                               // To send data over the ATM
CEvaluate SchedEval[3];                         // Evaluate timer tick, looping time + actual time to process
CEvaluate NotAccessible;

CScheduler::CScheduler()                        // Constructor - initialise variables
{counter = 0;
int loop;
ScheduleEvent = new CEvent();
for( loop=0; loop<MAXCLIENTS; loop++ )
{Movies[loop].VideoActive = FALSE;
Movies[loop].BufferOne.Accessible = FALSE;
Movies[loop].BufferOne.LastBuffer = FALSE;
Movies[loop].BufferOne.Start = NULL;
Movies[loop].BufferOne.End = NULL;
Movies[loop].BufferOne.ABS_End = NULL;
Movies[loop].BufferOne.Current_Pos = NULL;
Movies[loop].BufferTwo.Accessible = FALSE;
Movies[loop].BufferTwo.Start = NULL;
Movies[loop].BufferTwo.End = NULL;
Movies[loop].BufferTwo.ABS_End = NULL;
Movies[loop].BufferTwo.Current_Pos = NULL;
Movies[loop].BufferTwo.LastBuffer = FALSE;
Movies[loop].BufferOneActive = TRUE; } }

CScheduler::~CScheduler()                       // Destructor - Removes the ScheduleEvent object from the heap
{delete ScheduleEvent; }

void CScheduler::Timer()
{UINT SleepTime = 33;
MMRESULT mmres = timeBeginPeriod( 1 );

while(1)
{DWORD startTime = timeGetTime();
ScheduleEvent-ResetEvent();
Sleep( SleepTime );
ScheduleEvent-SetEvent();
DWORD endTime = timeGetTime();
if( endTime - startTime  SleepTime )
SleepTime = 33 - (( endTime - startTime ) - SleepTime );
else
SleepTime = 33; }
timeEndPeriod( 1 ); }

void CScheduler::DoSchedule()                   // Main thread - Scheduling

{DWORD startTime1;
DWORD startTime2;
DWORD atmtime;
DWORD endTime;
```

```
int loop;
SetBufMax(0);
SchedEval[0].OpenFile("ATM Tx times.txt");
SchedEval[1].OpenFile("Scheduler times.txt");
SchedEval[2].OpenFile("Actual Scheduler times.txt");
NotAccessible.OpenFile("Not_Accessible.txt");
MMRESULT mmres = timeBeginPeriod( 1 );
Quit = FALSE;
Sleep(1000);
int SleepTime = 33;
int PivotTime = 33;
double loops = 0.0;
double time = 0.0;
double pivot;
DWORD framecount = 0;

while(!Quit)                                    // Main loop
{startTime1 = timeGetTime();
if( SleepTime  0 )
Sleep( SleepTime );
startTime2 = timeGetTime();
for( loop=0; loop<=BufMax; loop++ )             // Loop through the clients
{
if( Movies[loop].VideoActive )                  // If the video is active
{
if((!Movies[loop].BufferOne.Accessible) &&
(!Movies[loop].BufferTwo.Accessible))
NotAccessible.SetData( framecount );
else
framecount++;
if(Movies[loop].BufferOneActive)                // If BufferOne is active

Movies[loop].BufferOne.Crit_Section.Lock();     // Lock the buffer's critical section
if(Movies[loop].BufferOne.Accessible)           // If the BufferOne is accessible - ie. full
{
// Check if packet size is not greater to the rest of the buffer to schedule out
 if( Movies[loop].BufferOne.End - Movies[loop].BufferOne.Current_Pos <
Movies[loop].BufferOne.PacketSize )
Movies[loop].BufferOne.PacketSize = Movies[loop].BufferOne.End - Movies[loop].BufferOne.Current_Pos;
// If so, reassign packet size
// Transmit the data over the ATM network
atmtime = timeGetTime();
ATM.TxSection( Movies[loop].BufferOne.PacketSize, Movies[loop].BufferOne.Current_Pos,Movies[loop].VideoSocket);
SchedEval[0].SetData( timeGetTime() - atmtime );
// Increment the Current position pointer of the buffer
Movies[loop].BufferOne.Current_Pos += Movies[loop].BufferOne.PacketSize;
Movies[loop].BufferOneActive = TRUE;
Movies[loop].BufferOne.Accessible = TRUE;
if( Movies[loop].BufferOne.Current_Pos = Movies[loop].BufferOne.End
{                                               // If buffer is empty
Movies[loop].BufferOneActive = FALSE;           // Switch to other buffer
Movies[loop].BufferOne.Accessible = FALSE;      // Mark buffer as empty
DM-AddABufferToList( GetBuffer(loop), loop );
if( Movies[loop].BufferOne.LastBuffer == TRUE || framecount = 67000 )
{
SetMovieInActive( loop );
Movies[loop].BufferOne.LastBuffer = FALSE;
ATM.SendFinished( Movies[loop].VideoSocket ); } } }
Movies[loop].BufferOne.Crit_Section.Unlock();   // Release the crit. section
}
else                                            // Buffer Two is active
{
Movies[loop].BufferTwo.Crit_Section.Lock();     // Lock the buffer's crit. section
if (Movies[loop].BufferTwo.Accessible)          // Check if buffer is accessible
{
// Check if packet size is not greater to the rest of the buffer to schedule out
if( Movies[loop].BufferTwo.End - Movies[loop].BufferTwo.Current_Pos <
Movies[loop].BufferTwo.PacketSize )
Movies[loop].BufferTwo.PacketSize = Movies[loop].BufferTwo.End - Movies[loop].BufferTwo.Current_Pos;
```

```
// If so, reassign packet size
// Transmit the data over the ATM network
atmtime = timeGetTime();
ATM.TxSection( Movies[loop].BufferTwo.PacketSize, Movies[loop].BufferTwo.Current_Pos,Movies[loop].VideoSocket );
SchedEval[0].SetData( timeGetTime() - atmtime );
// Increment the Current position pointer of the buffer
Movies[loop].BufferTwo.Current_Pos += Movies[loop].BufferTwo.PacketSize;
if( Movies[loop].BufferTwo.Current_Pos = Movies[loop].BufferTwo.End )
{                                                    // If buffer is empty.
Movies[loop].BufferOneActive = TRUE;                 // Switch to other buffer
Movies[loop].BufferTwo.Accessible = FALSE;           // Mark buffer as empty
DM-AddABufferToList( GetBuffer(loop), loop );
if( Movies[loop].BufferTwo.LastBuffer == TRUE || framecount = 67000 )
{
SetMovieInActive( loop );
Movies[loop].BufferTwo.LastBuffer = FALSE;
ATM.SendFinished( Movies[loop].VideoSocket ); } } }
Movies[loop].BufferTwo.Crit_Section.Unlock(); } } }  // Unlock the crit. section
endTime = timeGetTime();
time += ( endTime - startTime1 );
loops++;
pivot = FRAMERATE + ( FRAMERATE - (time/loops));
PivotTime = pivot + 0.5;
SleepTime = PivotTime - ( endTime - startTime2 );
SchedEval[1].SetData( endTime - startTime1 );
SchedEval[2].SetData( endTime - startTime2 ); }
timeEndPeriod( 1 ); }

BUFFERINFO * CScheduler::GetBuffer(int BufNum)       // Method to get the inactive buffer
{
if(Movies[BufNum].BufferOneActive)                   // If Buffer One is active,
return &Movies[BufNum].BufferTwo;                    // return buffer two's address.
else                                                 // Otherwise Buffer two is active
return &Movies[BufNum].BufferOne;                    // so return buffer one's address
}

// Method set up a client's buffers
BUFFERINFO * CScheduler::SetUpBuffers( int BufNum, int MaxPacket, SOCKET control_socket, SOCKET video_socket)
{
if( (Movies[BufNum].BufferOne.Start = new BYTE[MaxPacket]) == NULL ) // Assign memory for buffer one
return NULL;
if( (Movies[BufNum].BufferTwo.Start = new BYTE[MaxPacket]) == NULL ) // Assign memory for buffer two

{ // If this fails,
delete Movies[BufNum].BufferOne.Start;               // Delete buffer one's memory
Movies[BufNum].BufferOne.Start = NULL;
return NULL;
}
if( ! VirtualLock( Movies[BufNum].BufferOne.Start, MaxPacket ) )
{DWORD error1 = GetLastError();
AfxMessageBox("Locking of BufferOne not successful!");
char out[10];
itoa( error1, out, 10 );
AfxMessageBox( out ); }
if( ! VirtualLock( Movies[BufNum].BufferTwo.Start, MaxPacket ) )
{DWORD error2 = GetLastError();
AfxMessageBox("Locking of BufferTwo not successful!");
char out2[10];
itoa( error2, out2, 10 );
AfxMessageBox( out2 ); }
// Buffer one setup
Movies[BufNum].BufferOne.ABS_End = Movies[BufNum].BufferOne.Start + MaxPacket; // Absolute end of the buffer
Movies[BufNum].BufferOne.End = Movies[BufNum].BufferOne.Start;                 // Empty
Movies[BufNum].BufferOne.Current_Pos = Movies[BufNum].BufferOne.Start;         // At beginning of buffer
Movies[BufNum].BufferOne.PacketSize = 0;
Movies[BufNum].BufferOne.Accessible = FALSE;
Movies[BufNum].BufferOne.LastBuffer = FALSE;

// Buffer two setup
```

```
Movies[BufNum].BufferTwo.ABS_End = Movies[BufNum].BufferTwo.Start + MaxPacket; // Absolute end of the buffer
Movies[BufNum].BufferTwo.End = Movies[BufNum].BufferTwo.Start;                  // Empty
Movies[BufNum].BufferTwo.Current_Pos = Movies[BufNum].BufferTwo.Start;          // At beginning of buffer
Movies[BufNum].BufferTwo.PacketSize = 0;
Movies[BufNum].BufferTwo.Accessible = FALSE;
Movies[BufNum].BufferTwo.LastBuffer = FALSE;

// Movie setup
Movies[BufNum].VideoActive = FALSE;
Movies[BufNum].BufferOneActive = TRUE;

// Link the client's sockets to this stream
Movies[BufNum].ControlSocket = control_socket;
Movies[BufNum].VideoSocket = video_socket;

return &Movies[BufNum].BufferOne;                    // Setup a success, return the address of the first buffer
}

void CScheduler::SetMovieActive( int BufNum )        // Allow the movie to be scheduled
{Movies[BufNum].VideoActive = TRUE;}

void CScheduler::SetMovieInActive( int BufNum )      // Stop the movie from scheduling
{Movies[BufNum].VideoActive = FALSE;}

// Remove the buffers from memory and set the variables back to unset
void CScheduler::RemoveBuffers( int BufNum )
{Movies[BufNum].VideoActive = FALSE;                 // Movie removal

if( ! VirtualUnlock( Movies[BufNum].BufferOne.Start,  // Buffer one removal
Movies[BufNum].BufferOne.ABS_End - Movies[BufNum].BufferOne.Start ) )
AfxMessageBox("Unlocking of BufferOne not successful!");
delete Movies[BufNum].BufferOne.Start;
Movies[BufNum].BufferOne.Start = NULL;
Movies[BufNum].BufferOne.ABS_End = NULL;
Movies[BufNum].BufferOne.End = NULL;
Movies[BufNum].BufferOne.Current_Pos = NULL;
Movies[BufNum].BufferOne.PacketSize = 0;

if( ! VirtualUnlock( Movies[BufNum].BufferTwo.Start,  // Buffer two removal
Movies[BufNum].BufferTwo.ABS_End - Movies[BufNum].BufferTwo.Start ) )
AfxMessageBox("Unlocking of BufferOne not successful!");
delete Movies[BufNum].BufferTwo.Start;
Movies[BufNum].BufferTwo.Start = NULL;
Movies[BufNum].BufferTwo.ABS_End = NULL;
Movies[BufNum].BufferTwo.End = NULL;
Movies[BufNum].BufferTwo.Current_Pos = NULL;
Movies[BufNum].BufferTwo.PacketSize = 0;
}

void CScheduler::SetQuit()                           // Sets quit to end the DoSchedule thread
{Quit = TRUE;}

void CScheduler::SetBufMax(BYTE BufMaxIn)
{BufMax = BufMaxIn;}

void CScheduler::SetDM(CDisk_Manager * DiskManager)
{DM = DiskManager;}
```

1.10 Scheduler.h

```
#include "afxmt.h"     // For CEvent class and threads
#include "Globals.h"   // BUFFERINFO, etc

class CDisk_Manager;
class CScheduler
{public:
CScheduler();
public:
```

```
void SetDM(CDisk_Manager * DiskManager);
void SetBufMax( BYTE BufMaxIn );
void Timer();
void SetQuit();                                  // Sets quit variable to close the DoSchedule thread
void DoSchedule();                               // The main thread to do the scheduling
void RemoveBuffers(int BufNum);                  // Removes the buffers of a client that have been allocated
void SetMovieInActive(int BufNum);               // Stops movie from scheduling
void SetMovieActive(int BufNum);                 // Starts movie scheduling
// Sets up buffers for client and returns a pointer to the first buffer
BUFFERINFO * SetUpBuffers(int BufNum, int MaxPacket, SOCKET control_socket, SOCKET video_socket);
// Returns the address of the buffer that is inactive
BUFFERINFO * GetBuffer( int BufNum );
virtual ~CScheduler();
private:
BYTE BufMax;
CDisk_Manager * DM;
CEvent * ScheduleEvent;                          // To use the CEvent class's timer
SCHEDULER_INFO Movies[20];                       // Array of the Movies with clients buffers and addresses, etc.
BOOL Quit;                                       // Set to end DoSchedule thread
int counter; };
```

1.11 Stream_Manager.cpp

```
#include "stdafx.h"
#include "ServerVOD.h"
#include "Stream_Manager.h"
#include "ListConnections.h"
#include "ATM.h"
#define SCHEDULER_BUFSIZE 1000000
extern CATM ATM;
extern ListConnections clients;
extern char * string;

CStream_Manager::CStream_Manager() {}
CStream_Manager::~CStream_Manager() {}
void CStream_Manager::InitSM( )                  // Initialise Stream Manager to run
{ HANDLE ThisProcess = GetCurrentProcess();
DWORD min, max;
if( ! GetProcessWorkingSetSize( ThisProcess, &min, &max ) )
AfxMessageBox("GetProcessWorkingSetSize didn't succeed");
min = 2500000;                                   // 2meg avail to lock at present
if( max < min )
max = min + 100000;
if( ! SetProcessWorkingSetSize( ThisProcess, min, max ) )
AfxMessageBox("SetProcessWorkingSetSize didn't succeed");
BufMax = 0;
Sched = new CScheduler();
DM = new CDisk_Manager( Sched );
Sched-SetDM( DM );
AfxBeginThread( StartSchedule, Sched, THREAD_PRIORITY_TIME_CRITICAL );   // Start the scheduler thread
AfxBeginThread( StartDM, DM, THREAD_PRIORITY_HIGHEST );}                 // Start the Disk Manager thread

UINT CStream_Manager::StartDM( LPVOID pParam )   // Function to start the Disk Manager ReadDM function in a
                                                 // thread
{CDisk_Manager * DManager;
DManager = (CDisk_Manager*)pParam;
DManager-ReadDM();
delete DManager;
return 0; }

UINT CStream_Manager::StartTimer( LPVOID pParam )
{CScheduler * Scheduler;
Scheduler = (CScheduler*)pParam;
Scheduler-Timer();
return 0; }

// Function to start the Scheduler DoSchedule function in a thread
UINT CStream_Manager::StartSchedule( LPVOID pParam )
```

```
{CScheduler * Scheduler;
Scheduler = (CScheduler*)pParam;
Scheduler-DoSchedule();
delete Scheduler;
return 0;}

// For ATM to allow stop, etc requests.  Called by ATM class
void CStream_Manager::DoSM(char * params, int bufnum, SOCKET control_socket, SOCKET video_socket)
{switch (params[0])
{case 'N' :  NewMovie(control_socket,video_socket,bufnum);
break;
case 'S' :  StopMovie(bufnum);
break;
case 'P' :  PlayMovie(bufnum);
break;
case 'C' :  CancelMovie(video_socket,bufnum);
break;
default :
AfxMessageBox("DoSM: Error receiving request type - wrong char.");
AfxMessageBox(params); } }

void CStream_Manager::NewMovie(SOCKET control_socket,SOCKET video_socket,int bufnum)
{char * MovieName = ATM.Receive_Movie_Choice(control_socket);
if(strcmp(MovieName,"Receive Error")==0)
{ AfxMessageBox("Invalid Movie Name"); }

else

{clients.Update_Video_by_Bufnum(MovieName, bufnum);
if(strcmp( MovieName,"cancelled" ) !=0 )
{ BUFFERINFO * FirstBuf;

// Set up buffers and get pointer to first buffer
FirstBuf = Sched-SetUpBuffers( bufnum, SCHEDULER_BUFSIZE, control_socket, video_socket);

if (FirstBuf == NULL)  // If buffers not set up
{AfxMessageBox("NewMovie: Insufficient Memory, video buffers not set up.");
exit(0); }
DM-AddToDM( bufnum, MovieName, FirstBuf );                  // Add a new movie stream to the DM
DM-AddABufferToList( Sched-GetBuffer( bufnum ), bufnum );  // Get the inactive buffer i.e. the second buffer

if( bufnum  BufMax )
{BufMax = bufnum;
Sched-SetBufMax( BufMax ); } } } }

void CStream_Manager::PlayMovie(int bufnum)                 // Let the video be scheduled out
{ Sched-SetMovieActive( bufnum ); }

void CStream_Manager::StopMovie(int bufnum)                 // Stop the movie stream from being scheduled out
{Sched-SetMovieInActive(bufnum);}

// Cancels a movie's stream
void CStream_Manager::CancelMovie(SOCKET video_socket,int bufnum)
{Sched-SetMovieInActive( bufnum );                          // Stop Scheduling it
DM-CloseStream( bufnum );                                   // Close its DM stream
Sched-RemoveBuffers( bufnum );                              // Remove sched. buffers
if( bufnum == BufMax )
{BufMax = ATM.NewBufMax( bufnum );
Sched-SetBufMax( BufMax ); }
ATM.SendFinished(video_socket); }

// Close the Stream Manager module and its related modules down
void CStream_Manager::CloseSM()
{  // Stops all movies scheduling and frees their buffers
for( int loop=0; loop<BufMax; loop++ )
{Sched-SetMovieInActive(loop);
Sched-RemoveBuffers(loop); }
DM-CloseAllFiles();                                         // Closes all the open video files
Sched-SetQuit();                                            // Ends Scheduler thread
```

```
DM-SetQuit();                                   // Ends the Disk Manager thread
delete Sched;
delete DM;}
```

1.12 Stream_Manager.h

```
#include "Scheduler.h"
#include "Disk_Manager.h"

class CStream_Manager
{public:
CStream_Manager();
public:
void DoSM( char * params,int bufnum, SOCKET sock1, SOCKET sock2);   // Function called by SM to adjust stream
void CancelMovie(SOCKET sock2, int bufnum);                         // Cancels Movie stream
void CloseSM();                                                     // Shuts down the Stream Manager and its
                                                                    // dependants
void StopMovie(int bufnum);                                         // Stops a video from being sent.
void PlayMovie(int bufnum);                                         // Allows a video to be sent
void InitSM();                                                      // Initialises the Stream Manager
void NewMovie(SOCKET sock1,SOCKET sock2,int bufnum);                // Adds a new stream to be scheduled
static UINT StartDM( LPVOID pParam );
static UINT StartSchedule( LPVOID pParam );                         // Function to start the Scheduler thread
static UINT StartTimer( LPVOID pParam );
virtual ~CStream_Manager();
private:
BYTE BufMax;
CScheduler * Sched;                                                 // Scheduler object
CDisk_Manager * DM;                                                 // Disk Manager object
HWND ErrorWindow;};
```

1.13 VideoDatabase.cpp

```
#include "stdafx.h"
#include "ServerVOD.h"
#include "VideoDatabase.h"
char mem_block[10000];                                              // A tunable parameter depending on how many
                                                                    // videos in future

VideoDatabase::VideoDatabase() {}
VideoDatabase::~VideoDatabase() {}

int VideoDatabase::Search_on_Video_Name(char* video_name)
{
// Assuming use of header node, find video_name
// If found, return TRUE and set the current position

Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)
{if(strcmp(P-video_name,video_name)==0)
{Current_Pos = P;
return 1; } };
return 0; };

char * VideoDatabase::Get_Server_Address(char* video_name)
{ char*message = "Not Found.";
Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)

{if(strcmp(P-video_name,video_name)==0)
{ Current_Pos = P;
return P-server_address; } };
return 0; }

int VideoDatabase::Search_on_Video_Type(char* video_type)
{
// Assuming use of header node, find video_name
```

// If found, return TRUE and set the current position

```
Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)
{if(strcmp(P-video_type,video_type)==0)
{Current_Pos = P;
return 1; } };
return 0; };

int VideoDatabase::Delete_by_Video_Name(char* video_name)
{ Node *Tmp;
for(Node *P = List_Head;P-Next != NULL; P=P-Next)
{ if(strcmp( (P-video_name),video_name) == 0)
break;
Tmp = P; }
Tmp-Next = P-Next;
delete P;
return 0;}

void VideoDatabase::Insert_In_Order(char *video_name,char *video_length,char *video_type, char *server_name, char
*server_address, char *broker_name, char *broker_address, char *description)
{for(Node *P = List_Head;P-Next != NULL; P=P-Next)
{if( strcmp((P-Next-video_name),video_name)  0)
break;}

Current_Pos = P;
Node *Tmp = new Node;
Tmp-video_name = video_name;
Tmp-video_length = video_length;
Tmp-video_type = video_type;
Tmp-server_name = server_name;
Tmp-server_address = server_address;
Tmp-broker_name = broker_name;
Tmp-broker_address = broker_address;
Tmp-description = description;
Tmp-Next = Current_Pos-Next;
Current_Pos-Next = Tmp;
Current_Pos = Current_Pos-Next; }

void VideoDatabase::Print_List()
{Node *P;
P = List_Head;
CString view;
while (P != NULL)
{view += CString(P-video_name) + "-----";
P = P-Next;}
view += "Null";
AfxMessageBox(view);};


CString VideoDatabase::Get_Next()
{Node *tmp = Current_Pos;
Current_Pos = Current_Pos-Next;
return (CString) tmp-video_name;};


void VideoDatabase::Set_Current(int index)
{int counter=0;
Current_Pos = List_Head-Next;
while(counter != index)
{  Current_Pos = Current_Pos-Next;
counter++; } };



int VideoDatabase::Empty()
{if(Current_Pos==NULL)
return 1;
else return 0; }
```

```
CString VideoDatabase::Get_Field(int field)
{CString result;
if(Current_Pos != NULL)
{switch(field)
{case 0:result = (CString) Current_Pos-video_name;break;
case 1:result = (CString) Current_Pos-video_length;break;
case 2:result = (CString) Current_Pos-video_type;break;
case 3:result = (CString) Current_Pos-server_name;break;
case 4:result = (CString) Current_Pos-server_address;break;
case 5:result = (CString) Current_Pos-broker_name;break;
case 6:result = (CString) Current_Pos-broker_address;break;
case 7:result = (CString) Current_Pos-description;break;
default:AfxMessageBox("Error in Get_Field");break; } };
return result; };


CString VideoDatabase::Save()
{CString save_string;
Set_Current(0);                                    // Go to the first video
while(!Empty())
{save_string += Current_Pos-video_name;
save_string += "\r";
save_string += Current_Pos-video_length;
save_string += "\r";
save_string += Current_Pos-video_type;
save_string += "\r";
save_string += Current_Pos-server_name;
save_string += "\r";
save_string += Current_Pos-server_address;
save_string += "\r";
save_string += Current_Pos-broker_name;
save_string += "\r";
save_string += Current_Pos-broker_address;
save_string += "\r";
save_string += Current_Pos-description;
save_string += "\r";
Current_Pos = Current_Pos-Next; };
save_string += "\f";

int store_size = save_string.GetLength();          // Save it to vidbase.txt
CFile videofile("vidbase.txt", CFile::modeCreate | CFile::modeReadWrite);
CArchive arOut(&videofile,CArchive::store,store_size);
arOut.WriteString(save_string);
arOut.Flush();
videofile.Close();
return save_string;};

CString VideoDatabase::Load()
{CString load_string;
CStdioFile VideoFile;
CString video,videos;

if (!VideoFile.Open("vidbase.txt", CFile::modeCreate | CFile::modeNoTruncate | CFile::modeReadWrite))
{AfxMessageBox("Error: Opening vidbase.txt");
return ""; }

else
{VideoFile.SeekToBegin();
while (VideoFile.ReadString(video))
{load_string += video; } }
VideoFile.Close();
return load_string; };

void VideoDatabase::Parse_Video_String(CString temp)
{
int i=0;
int counter = 0;
```

```
char *ptr1,*ptr2,*ptr3,*ptr4,*ptr5,*ptr6,*ptr7,*ptr8;
ptr1 = mem_block;
while(temp[i] != '\f')
{while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr2=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr3=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr4=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr5=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr6=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr7=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr8=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;
Insert_In_Order(ptr1,ptr2,ptr3,ptr4,ptr5,ptr6,ptr7,ptr8);
ptr1 = &mem_block[i]; } };

CString VideoDatabase::Create_Video_List_For_Broker()
{ CString the_list;
Set_Current(0);                                    // Go to the first video
while(!Empty())
{the_list += Current_Pos-video_name;
the_list += "\r";
the_list += Current_Pos-video_length;
the_list += "\r";
the_list += Current_Pos-video_type;
the_list += "\r";
the_list += Current_Pos-server_name;
the_list += "\r";
the_list += Current_Pos-server_address;
the_list += "\r";
the_list += Current_Pos-broker_name;
the_list += "\r";
the_list += Current_Pos-broker_address;
the_list += "\r";
the_list += Current_Pos-description;
the_list += "\r";
Current_Pos = Current_Pos-Next; };
the_list += "\n";
return the_list; };

void VideoDatabase::Initialize_List()
{List_Head = new Node();
List_Head-video_name="header";
List_Head-video_length="0";
List_Head-video_type="header";
List_Head-Next = NULL;
Current_Pos = new Node();
Current_Pos = List_Head;
};
```

1.14 VideoDatabase.h

```
class VideoDatabase : public CFrameWnd
{

public:
VideoDatabase();
~VideoDatabase();
struct Node

{char *video_name;                                 // Can be 32 character's long
char *video_length;                                // Target Format: 00:00:00
char *video_type;                                  // Entertainment, Comedy, Documentary, Drama, Action
                                                   // News, Actuality, Sport

char *server_name;
char *server_address;
char *broker_name;
char *broker_address;
char *description;
Node *Next;
```

```
Node() {}    };                                    // Constructor for Node

Node *List_Head;
Node *Current_Pos;
void Insert_In_Order(char *video_name,char *video_length,char *video_type, char *server_name, char *server_address, char
*broker_name, char *broker_address, char *description);
int Search_on_Video_Name(char* video_name);
char * Get_Server_Address(char* video_name);
int Search_on_Video_Type(char* video_name);
int Delete_by_Video_Name(char* video_name);
void Print_List();
void Initialize_List();
CString Get_Next();
int Empty();
void Set_Current(int index);
CString Get_Field(int field);
CString Save();
CString Load();
void Parse_Video_String(CString temp);
CString Create_Video_List_For_Broker();
protected: };
```

2. Broker Classes

2.1 ATM.cpp

```
#include "stdafx.h"
#include "BrokerVOD.h"
#include "ATM.h"
#include "uuid.h"
#include "process.h"
#include "ListConnections.h"
#include "VideoDatabase.h"
extern ListConnections Brokers;
extern ListConnections Servers;
extern ListConnections Clients;
extern VideoDatabase Video_List;

ATM::ATM() {}
ATM::~ATM() {}

void ATM::Launch()
{WORD ver_req = MAKEWORD(2, 0);                    // Initialize the Winsock 2.0 Library
WSADATA wsa_data;

if (WSAStartup(ver_req, &wsa_data) != 0)
{AfxMessageBox("Error : Starting up WinSock2.0 DLL");
WSACleanup();
exit(0);}

if (LOBYTE(wsa_data.wVersion) != 2 || HIBYTE(wsa_data.wVersion) != 0)
{AfxMessageBox("Error : Could not find WinSock2.0 DLL");
WSACleanup();
exit(0);}
AfxBeginThread(ListenToConnectionSocket,NULL,THREAD_PRIORITY_NORMAL);
};

/*****************************************************************/
/* Function Name:     ListenToConnectionSocket                   */
/* Function:          Listens for client requests to the Spawn Socket,  */
/*                    when a new client wants to,join it creates a unique */
/*                    socket and spins the thread.               */
/*****************************************************************/


UINT ATM::ListenToConnectionSocket(LPVOID param)
```

```
{  SOCKET SpawnSocket;

// Create a spawning socket (that we will listen on)
if ((SpawnSocket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{  AfxMessageBox("WSASocket - Error Creating SpawnSocket");
WSACleanup();
exit(0); }

// Get this broker's address from the underlying Winsock info

struct sockaddr_atm my_addr;
ULONG bytes = 0;
DWORD deviceID = 0;
if(WSAIoctl(SpawnSocket,SIO_GET_ATM_ADDRESS,(LPVOID)&deviceID,sizeof(deviceID),(LPVOID)&my_addr,sizeof(my_a
ddr),&bytes,NULL,NULL) == SOCKET_ERROR)
AfxMessageBox("Error getting my ATM address.");

// A fix for the 4-byte offset problem

struct sockaddr_atm sock_addr;

for(int l = -4; l < 16; l ++ )
{  sock_addr.satm_number.Addr[l+4] = my_addr.satm_number.Addr[l];};

// Go to the atmhosts file, compare this address until we find a match and then we have the machine name of this broker.

sock_addr.satm_family          = AF_ATM;                  // Address Family is ATM
sock_addr.satm_number.AddressType = ATM_NSAP;             // Address Style is network service access point
sock_addr.satm_number.NumofDigits = ATM_ADDR_SIZE;        // ATM_ADDR_SIZE = 20 (defined in ws2atm.h)
sock_addr.satm_blli.Layer2Protocol = SAP_FIELD_ANY;       // Defined in ws2atm.h
sock_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;    // Defined in ws2atm.h
sock_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT; // Defined in ws2atm.h

// Make sure we don't interfere with LANE / CLIP selector bytes

sock_addr.satm_number.Addr[19] = 0x80;

// At this point in the code, we have a socket (called "SpawnSocket") which is about to be bound to the address we have placed in
// the sockaddr_atm structure called sock_addr. Bind the broker ATM address to the Socket created by WSASocket()

if (bind(SpawnSocket, (struct sockaddr FAR *)&sock_addr, sizeof(struct sockaddr_atm))== SOCKET_ERROR)
{AfxMessageBox("Error : Bind Spawnsocket");
exit(0); }

struct sockaddr_atm ClientAddr;                           // The structure which holds the client's address information
ATM_CONNECTION_ID connID;                                 // The structure which holds DeviceNumber,VPI,VCI info
int len;
int sel=0;
SOCKET NewCustomerSocket;

while(1)

{if(listen(SpawnSocket, 5) == SOCKET_ERROR)               // Listen for a client request to connect
{AfxMessageBox("Error : listen() on Spawnsocket");
break;}

// Accept the client connection request

if ((NewCustomerSocket = WSAAccept(SpawnSocket,(struct sockaddr FAR *)&ClientAddr,&len,
NULL,0)) == SOCKET_ERROR)
{AfxMessageBox("Error : accept");}

// WSAIoctl: This routine is used to set or retrieve operating parameters associated with
// the socket, the transport protocol, or the communications subsystem


if (WSAIoctl(NewCustomerSocket,SIO_GET_ATM_CONNECTION_ID,NULL,0,(LPVOID) &connID,
sizeof(ATM_CONNECTION_ID),&bytes,NULL,NULL)== SOCKET_ERROR)
{AfxMessageBox("Error: WSAIoctl");}
```

```
// Receive an identifier from the connected entity stating whether it is a Broker, Server or Client

WSABUF id;
id.len = 1;
id.buf = (char *) malloc(1);
ULONG b,f;
if((WSARecv(NewCustomerSocket,&id,1, &b, &f, NULL, NULL)) == SOCKET_ERROR)
{ int y=WSAGetLastError();
char * error;
_itoa(y,error,10);
AfxMessageBox(error);
};
CString temp = id.buf;
temp.GetBufferSetLength(1);
Transfer Info;
Info.ThisCustomerSocket = NewCustomerSocket;
Info.DeviceNumber = connID.DeviceNumber;
Info.VCI = connID.VCI;
Info.VPI = connID.VPI;

// Pass this client to a Comms thread - either Broker, Server or Client

if(strcmp(temp,"B")==0)   AfxBeginThread(BrokerComms,(LPVOID)&Info,THREAD_PRIORITY_NORMAL);
else
if(strcmp(temp,"S")==0)   AfxBeginThread(ServerComms,(LPVOID)&Info,THREAD_PRIORITY_NORMAL);
else
if(strcmp(temp,"C")==0) AfxBeginThread(ClientComms,(LPVOID)&Info,THREAD_PRIORITY_NORMAL);
}; return 0;}

/*******************************************************/
/* Function Name:      ClientComms                     */
/* Function:           Communicates with a unique client  */
/*******************************************************/

UINT ATM::ClientComms(LPVOID param)

{  Transfer *p;
p =(Transfer*) param;
SOCKET client_socket = p-ThisCustomerSocket;

// It is better to fill in the Client Connection details here than in the listen loop, because it frees up the listen thread.
// Get the Client's Name

CString client_name;
WSABUF Receive;
Receive.len = 10;
Receive.buf = (char *) malloc(10);
ULONG Bytes = 0;
ULONG Flags = 0;
WSARecv(client_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
client_name = Receive.buf;
client_name.GetBufferSetLength(10);

CString client_address;                          // Get the Client's address in neat, printable format
Receive.len = 40;
Receive.buf = (char *) malloc(40);
WSARecv(client_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
client_address = Receive.buf;
client_address.GetBufferSetLength(40);

CString temp_copy = client_address;              // Create an ATM-style char address for this client.
char client_addr[20];
int index = 0;
int i=0;
int m=0;
char j;
char k;
while(i!=40)
```

```
{j = GetBinary(temp_copy[i]);
k =  GetBinary(temp_copy[i+1]);
j <<= 4;
client_addr[m] = j | k;
i += 2;
m++; };

CTime time = CTime::GetCurrentTime();                    // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

// Now add this new client to the Clients linked list
Clients.Insert_In_Order(client_name,client_address,connection_time);

// At this point we have the socket for this client until the client is finished. We need a loop that gets a request from the client and
// deals with it. Possible requests: (1) I want to select / search for a new movie (2) Disconnect from broker

Receive.len = 1;
Receive.buf = (char *) malloc(1);
while(1)
{if (WSARecv(client_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL) == SOCKET_ERROR)
{Clients.Delete_by_Name(client_name);                    // Remove client from linked list
closesocket(client_socket);                              // Close the client's socket
break;}
if(Receive.buf[0]=='G') Send_Videos_To_Client(client_socket);
if(Receive.buf[0]=='Q') QOS_Engine(client_socket); }
return 0; }

/*****************************************************************/
/* Function Name:      QOS_Engine                                */
/* Function:           Negotiates QOS with client for a video    */
/*****************************************************************/

void ATM::QOS_Engine(SOCKET client_socket)
{
WSABUF choice;                                           // Receive the video name from the client
choice.len = 50;
choice.buf = (char *) malloc(50);
ULONG choiceBytes = 0;
ULONG choiceFlags = 0;
if (WSARecv(client_socket, &choice, 1, &choiceBytes, &choiceFlags, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error receiving choice in QOS_Engine");}

// Here is where the negotiation will take place
// Find the server's address, given the video name

char *server_address;
server_address = Video_List.Get_Server_Address(choice.buf);

WSABUF address;                                          // Send the server's address to the client
address.len=40;
address.buf=(char *) malloc(40);
ULONG xmitBytes = 0;
ULONG Flags = 0;
address.buf = server_address;
if(WSASend(client_socket, &address, 1, &xmitBytes, Flags, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error transmitting server address to client",MB_OK,0);} };

/*****************************************************/
/* Function Name:      BrokerComms                   */
/* Function:           Communicates with a unique Broker */
/*****************************************************/

UINT ATM::BrokerComms(LPVOID param)
{Transfer *p;
p =(Transfer*) param;
SOCKET broker_socket = p-ThisCustomerSocket;

CString broker_name;                                     // Get the Brokers Name
WSABUF Receive;
```

```
Receive.len = 10;
Receive.buf = (char *) malloc(10);
ULONG Bytes = 0;
ULONG Flags = 0;
WSARecv(broker_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
broker_name = Receive.buf;
broker_name.GetBufferSetLength(10);
CString broker_address;                                  // Get the Broker's address in neat, printable format
Receive.len = 40;
Receive.buf = (char *) malloc(40);
WSARecv(broker_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
broker_address = Receive.buf;
broker_address.GetBufferSetLength(40);
return 0;
char broker_addr[20];
CTime time = CTime::GetCurrentTime();                    // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");
Brokers.Insert_In_Order(broker_name,broker_address,connection_time);      // Now add this new client to the Clients linked list
}

/*******************************************************/
/* Function Name:      ServerComms                     */
/* Function:           Communicates with a unique Server  */
/*******************************************************/

UINT ATM::ServerComms(LPVOID param)
{Transfer *p;
p =(Transfer*) param;
SOCKET server_socket = p-ThisCustomerSocket;
CString server_name;                                     // Get the Server's Name
WSABUF Receive;
Receive.len = 10;
Receive.buf = (char *) malloc(10);
ULONG Bytes = 0;
ULONG Flags = 0;
WSARecv(server_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
server_name = Receive.buf;
server_name.GetBufferSetLength(10);
CString server_address;                                  // Get the Server's address in neat, printable format
Receive.len = 40;
Receive.buf = (char *) malloc(40);
WSARecv(server_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL);
server_address = Receive.buf;
server_address.GetBufferSetLength(40);
CString temp_copy = server_address;                      // Create an ATM-style char address for this server
char server_addr[20];
int index = 0;
int i=0;
int m=0;
char j;
char k;
while(i!=40)
{j = GetBinary(temp_copy[i]);
k =  GetBinary(temp_copy[i+1]);
j <<= 4;
server_addr[m] = j | k;
i += 2;
m++;};

CString VidBase;                                         // Get the video database
Receive.len = 10000;
Receive.buf = (char *) malloc(10000);
if((WSARecv(server_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL)) == SOCKET_ERROR)
AfxMessageBox("Error");

VidBase = Receive.buf;
VidBase.GetBufferSetLength(10000);
Video_List.Parse_Video_String(VidBase);
CTime time = CTime::GetCurrentTime();                    // Get the connection time
```

```
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

// Now add this new server to the servers linked list

Servers.Insert_In_Order(server_name,server_address,connection_time);

// At this point we have the socket for this server until the server is finished.We need a loop that gets a request from the client and
// deals with it

Receive.len = 1;
Receive.buf = (char *) malloc(1);
while(1)
{if (WSARecv(server_socket, &Receive, 1, &Bytes, &Flags, NULL, NULL) == SOCKET_ERROR)
{Servers.Delete_by_Name(server_name);                    // Remove client from linked list
Video_List.Delete_by_Server(server_name);
closesocket(server_socket);                              // Close the server's socket
break; } }
return 0; }


/*******************************************************/
/* Function Name:      Send_Videos_To_Client           */
/* Function:           To send the list of movies to the client */
/*******************************************************/

void ATM::Send_Videos_To_Client(SOCKET client_socket)
{WSABUF Videos;
Videos.len=10000;
Videos.buf=(char *) malloc(10000);
ULONG xmitBytes = 0;
ULONG Flags = 0;
CString VideoList = Video_List.Create_Video_List_For_Client();
char *sender = VideoList.GetBuffer(10000);
Videos.buf = sender;
if(WSASend(client_socket, &Videos, 1, &xmitBytes, Flags, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error transmitting video list to client",MB_OK,0);} }

/*******************************************************/
/* Function Name:      SendFinished                    */
/* Function:           Stop the Client from plaiyng    */
/*******************************************************/

void ATM::SendFinished(SOCKET video_socket)
{  WSABUF Finished;
Finished.len=20;
Finished.buf=(char *) malloc(20);
ULONG xmitBytes = 0;
ULONG Flags = 0;
Finished.buf = "4";
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed");
Finished.len=4;
BYTE data[4] = {0x00, 0x00, 0x01, 0xB9};
char * dataptr = (char*)data;
Finished.buf = dataptr;
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed");
Finished.len=20;
Finished.buf = "Finished";
if(WSASend(video_socket,&Finished,1,&xmitBytes,Flags,NULL,NULL)!=0)
AfxMessageBox("Sending Finish Message Failed");}

char ATM::GetBinary(char the_char)
{char the_return;
switch(the_char)
{case '0':  the_return = 0x0;break;
case '1':  the_return = 0x1;break;
case '2':  the_return = 0x2;break;
case '3':  the_return = 0x3;break;
```

```
case '4':  the_return = 0x4;break;
case '5':  the_return = 0x5;break;
case '6':  the_return = 0x6;break;
case '7':  the_return = 0x7;break;
case '8':  the_return = 0x8;break;
case '9':  the_return = 0x9;break;
case 'a':  the_return = 0xa;break;
case 'b':  the_return = 0xb;break;
case 'c':  the_return = 0xc;break;
case 'd':  the_return = 0xd;break;
case 'e':  the_return = 0xe;break;
case 'f':  the_return = 0xf;break;
default: AfxMessageBox("Error in GetBinary()'s case statement"); }
return the_return; };
```

2.2 ATM.h

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <ws2atm.h>
#include <ws2spi.h>
#include <signal.h>
#include "Globals.h"                                    // For ClientConnections struct
#define FORE_NAME_SPACE NS_ALL

class ATM : public CFrameWnd
{protected:
public:
ATM();
virtual ~ATM();
void Launch();
static void Send_Videos_To_Client(SOCKET control_socket);
void SendFinished(SOCKET video_socket);
static void QOS_Engine(SOCKET client_socket);

struct ServerProperties
{CString name;
CString ATM_Address;
CString Socket_ID;
CString Address_Family;
CString Address_Style;
CString AAL;
CString  Socket_Style;
} Server;

struct Transfer

{ SOCKET ThisCustomerSocket;
DWORD DeviceNumber;
DWORD VPI;
DWORD VCI;
};
public:
static UINT ListenToConnectionSocket(LPVOID param);
static UINT BrokerComms(LPVOID param);
static UINT ServerComms(LPVOID param);
static UINT ClientComms(LPVOID param);
static char GetBinary(char the_char);
protected:
private:
static BOOL get_atm_address( char *, struct sockaddr_atm * );};
```

2.3 VideoDatabase.cpp

```
#include "stdafx.h"
```

```
#include "BrokerVOD.h"
#include "VideoDatabase.h"
char mem_block[10000];                    // A tunable parameter depending on how many videos in future

VideoDatabase::VideoDatabase() {}
VideoDatabase::~VideoDatabase() {}

int VideoDatabase::Search_On_Video_Name(char* video_name)
{

// Assuming use of header node, find video_name. If found, return TRUE and set the current position

Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)
{if(strcmp(P-video_name,video_name)==0)
{Current_Pos = P;
return 1;}
};
return 0;};

char * VideoDatabase::Get_Server_Address(char* video_name)
{char*message = "Not Found.";
Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)

{if(strcmp(P-video_name,video_name)==0)
{Current_Pos = P;
return P-server_address;} };
return 0; }

int VideoDatabase::Search_On_Video_Type(char* video_type)
{

// Assuming use of header node, find video_name. If found, return TRUE and set the current position

Node *P;
for(P=List_Head-Next;P != NULL; P=P-Next)
{if(strcmp(P-video_type,video_type)==0)
{Current_Pos = P;
return 1;
}};
return 0;};

int VideoDatabase::Delete_By_Video_Name(char* video_name)
{Node *Tmp;
for(Node *P = List_Head;P-Next != NULL; P=P-Next)
{if(strcmp( (P-video_name),video_name) == 0)
break;
Tmp = P;}
Tmp-Next = P-Next;
delete P;
return 0;}

void VideoDatabase::Delete_by_Server(CString server_name)
{char *server = server_name.GetBuffer(0);
Node *P;
Node *tmp;
P=List_Head-Next;
while(P != NULL)
{if(strcmp(P-server_name,server)==0)
{  tmp = P-Next;
Delete_By_Video_Name(P-video_name); }
P=tmp; } };

void VideoDatabase::Insert_In_Order(char *video_name,char *video_length,char *video_type, char *server_name, char
*server_address, char *broker_name, char *broker_address, char *description)

{for(Node *P = List_Head;P-Next != NULL; P=P-Next)
{if( strcmp((P-Next-video_name),video_name)  0)
```

```
break;}

Current_Pos = P;
Node *Tmp = new Node;
Tmp-video_name = video_name;
Tmp-video_length = video_length;
Tmp-video_type = video_type;
Tmp-server_name = server_name;
Tmp-server_address = server_address;
Tmp-broker_name = broker_name;
Tmp-broker_address = broker_address;
Tmp-description = description;
Tmp-Next = Current_Pos-Next;
Current_Pos-Next = Tmp;
Current_Pos = Current_Pos-Next; }

void VideoDatabase::Print_List()
{ AfxMessageBox("Printing List...");
Node *P;
P = List_Head;
CString view;
while (P != NULL)
{view += CString(P-video_name) + "-----";
P = P-Next;
}
view += "Null";
AfxMessageBox(view); };

CString VideoDatabase::Get_Next()
{Node *tmp = Current_Pos;
Current_Pos = Current_Pos-Next;
return (CString) tmp-video_name; };

void VideoDatabase::Set_Current(int index)
{ int counter=0;
Current_Pos = List_Head-Next;
while(counter != index)
{   Current_Pos = Current_Pos-Next;
counter++; } };

int VideoDatabase::Empty()
{ if(Current_Pos==NULL)
return 1;
else return 0; }

CString VideoDatabase::Get_Field(int field)
{CString result;

if(Current_Pos != NULL)
{ switch(field)
{ case 0:result = (CString) Current_Pos-video_name;break;
case 1:result = (CString) Current_Pos-video_length;break;
case 2:result = (CString) Current_Pos-video_type;break;
case 3:result = (CString) Current_Pos-server_name;break;
case 4:result = (CString) Current_Pos-server_address;break;
case 5:result = (CString) Current_Pos-broker_name;break;
case 6:result = (CString) Current_Pos-broker_address;break;
case 7:result = (CString) Current_Pos-description;break;
default:AfxMessageBox("Error in Get_Field");break;
} };

return result; };

CString VideoDatabase::Save()
{ CString save_string;
Set_Current(0);                                    // Go to the first video

while(!Empty())
{save_string += Current_Pos-video_name;
```

```
save_string += "\r";
save_string += Current_Pos-video_length;
save_string += "\r";
save_string += Current_Pos-video_type;
save_string += "\r";
save_string += Current_Pos-server_name;
save_string += "\r";
save_string += Current_Pos-server_address;
save_string += "\r";
save_string += Current_Pos-broker_name;
save_string += "\r";
save_string += Current_Pos-broker_address;
save_string += "\r";
save_string += Current_Pos-description;
save_string += "\r";
Current_Pos = Current_Pos-Next; };
save_string += "\f";

// Save it to vidbase.txt

int store_size = save_string.GetLength();
CFile videofile("vidbase.txt", CFile::modeCreate | CFile::modeReadWrite);
CArchive arOut(&videofile,CArchive::store,store_size);
arOut.WriteString(save_string);
arOut.Flush();
videofile.Close();
return save_string; };

CString VideoDatabase::Load()
{CString load_string;
CStdioFile VideoFile;
CString video, videos;
if (!VideoFile.Open("vidbase.txt", CFile::modeCreate | CFile::modeNoTruncate  | CFile::modeReadWrite))
{ AfxMessageBox("Error: Opening vidbase.txt");
return ""; }

else
{VideoFile.SeekToBegin();
while (VideoFile.ReadString(video))
{ load_string += video; } }
VideoFile.Close();
return load_string; };

void VideoDatabase::Parse_Video_String(CString temp)
{ int i=0;
int counter = 0;
char *ptr1,*ptr2,*ptr3,*ptr4,*ptr5,*ptr6,*ptr7,*ptr8;
ptr1 = mem_block;
while(temp[i] != '\n')
{ while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr2=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr3=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr4=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr5=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr6=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr7=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;ptr8=&mem_block[i];
while(temp[i]!='\r'){mem_block[i]=temp[i];i++;};mem_block[i] = '\0';i++;
Insert_In_Order(ptr1,ptr2,ptr3,ptr4,ptr5,ptr6,ptr7,ptr8);
ptr1 = &mem_block[i]; } };

CString VideoDatabase::Create_Video_List_For_Client()
{ CString the_list;
Set_Current(0);                                    // Go to the first video

while(!Empty())

{the_list += Current_Pos-video_name;
the_list += "\r";
the_list += Current_Pos-video_length;
```

```
the_list += "\r";
the_list += Current_Pos-video_type;
the_list += "\r";
the_list += Current_Pos-description;
the_list += "\r";
Current_Pos = Current_Pos-Next; };
the_list += "\n";
return the_list; };

void VideoDatabase::Initialize_List()
{List_Head = new Node();
List_Head-video_name="header";
List_Head-video_length="0";
List_Head-video_type="header";
List_Head-Next = NULL;
Current_Pos = new Node();
Current_Pos = List_Head; };
```

2.4 VideoDatabase.h

```
class VideoDatabase : public CFrameWnd
{public:
VideoDatabase();
~VideoDatabase();

struct Node
{char *video_name;                    // Can be 32 character's long
char *video_length;                   // Target Format: 00:00:00
char *video_type;                     // Entertainment, Comedy, Documentary, Drama, Action
                                      // News, Actuality, Sport

char *server_name;
char *server_address;
char *broker_name;
char *broker_address;
char *description;
Node *Next;
Node() {} };

Node *List_Head;
Node *Current_Pos;
void Insert(char* video_name,int video_length,char* video_type);
void Insert_In_Order(char *video_name,char *video_length,char *video_type, char *server_name, char *server_address, char
*broker_name, char *broker_address, char *description);
int Find(const char* X);
int Find_Previous(const char* X);
int Search_On_Video_Name(char* video_name);
char * Get_Server_Address(char* video_name);
int Search_On_Video_Type(char* video_name);
int Delete_By_Video_Name(char* video_name);
void Delete_by_Server(CString server_name);
void Print_List();
void Initialize_List();
CString Get_Next();
int Empty();
void Set_Current(int index);
CString Get_Field(int field);
CString Save();
CString Load();
void Parse_Video_String(CString temp);
CString Create_Video_List_For_Client();
protected: };
```

3. Client Classes

3.1 ATM.cpp

```
#include "stdafx.h"
#include "ClientVOD.h"
#include "ATM.h"
#include "uuid.h"
#include "SelectVideo.h"
#include "signal.h"
extern CBuffer Buffer;
extern CStringArray vids_name;
extern CStringArray vids_category;
extern CStringArray vids_time;
extern CStringArray vids_description;
CString videolist;
char mem_block[10000];
CFile videofile("data_log.txt",CFile::modeCreate | CFile::modeReadWrite);          // Open a File for data rate logging
char rate[10];

CATM::CATM()  {  video_end = new CEvent(); video_end-ResetEvent();  }
CATM::~CATM()  { delete video_end;}

/*******************************************************/
/* Function Name:      Init_Winsock                    */
/* Function:           To intialize the ATM stuff      */
/*******************************************************/

int CATM::Init_Winsock(CString broker_name)

{

// Steps:   (1)      Initialize Winsock2.
//          (2)      Create the client sockets:- control_socket,setup_video_socket,
//                   broker_socket
//          (3)      Get the client name, client address from the system and convert client address
//                   to CString format
//          (4)      Connect to the broker with broker_socket
//                   Use Fore Name Space Provider.

WORD ver_req = MAKEWORD(2, 0);
WSADATA wsa_data;

if (WSAStartup(ver_req, &wsa_data) != 0)
{AfxMessageBox("Error : Starting up WinSock2.0 DLL");}

if (LOBYTE(wsa_data.wVersion) != 2 || HIBYTE(wsa_data.wVersion) != 0)
{AfxMessageBox("Error : Could not find WinSock2.0 DLL");
WSACleanup();}

if ((broker_socket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{ AfxMessageBox("Unable to create broker_socket.\n");
WSACleanup();                                                    // Terminate the use of the WinSock2 DLL
exit(0);}

if ((control_socket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{  AfxMessageBox("Unable to create control_socket.\n");
WSACleanup();                                                    // Terminate the use of the WinSock2 DLL
exit(0);}


if ((setup_video_socket = WSASocket(AF_ATM, SOCK_RAW, ATMPROTO_AAL5, NULL, 0, 0)) == INVALID_SOCKET)
{  AfxMessageBox("Unable to create setup_video_socket.\n");
WSACleanup();                                                    // Terminate the use of the WinSock2 DLL
exit(0);}


// Get this client's address

DWORD bytes = 0;
DWORD deviceID = 0;
if(WSAIoctl(broker_socket,SIO_GET_ATM_ADDRESS,(LPVOID)&deviceID,
sizeof(deviceID),(LPVOID)&my_address,sizeof(my_address),&bytes,NULL,NULL) == SOCKET_ERROR)
```

```
AfxMessageBox("Error getting my ATM address.");

// Go to the atmhosts file, compare this address until we find a match and then we have the machine name of this client

client_name = Get_Name_By_Address(my_address,4);

// Convert address to Cstring. Code to get a "printable" ATM address. The crux is the conversion from unsigned char to hex with
// the sprintf. The format string ensures that leading zeroes are added where necessary for "nice"output

char next_byte[2];
UCHAR t;
for(int l = -4; l < 16; l ++ )
{ t = my_address.satm_number.Addr[l];
sprintf(next_byte,"%02x",(char*)t);
client_address += next_byte;};

char *broker = broker_name.GetBuffer(20);                    // Connect to the broker
if (Broker_Connect(broker_socket,broker) == FALSE)
{  WSACleanup();
return 0;}
else return 1; }

/*****************************************************************/
/* Function Name:      GetVideo                                  */
/* Function:           To receive video data sent by the server. */
/*****************************************************************/

void CATM::GetVideo()
{

// We are connected to the server. The following occurs: We have a loop in which we first receive the size of the next block of
// video data and then the actual block

Stop = FALSE;
ULONG xmitBytes = 0;
ULONG recvBytes = 0;
ULONG flags2 = 0;
int count;
WSABUF sizebuf;
ULONG sizeBytes = 0;
ULONG flags3 = 0;
int size;
int loop;
int y;
bool End;
int summer=0;
bool first = TRUE;
int start_time=0, end_time=0, data_rate=0;
while(!End)

{sizebuf.len = 20;                                           // First receive the size of the block of video data
sizebuf.buf = (char *) malloc(20);

if (WSARecv(video_socket, &sizebuf, 1, &sizeBytes, &flags3, NULL, NULL) == SOCKET_ERROR)
{  AfxMessageBox("Size");
char buffer[20];
_itoa( summer, buffer, 10 );
break;}

if(strcmp("4",sizebuf.buf)==0)
int i=0;


if(first && (strcmp("4",sizebuf.buf)==0))
{WSARecv(video_socket, &sizebuf, 1, &sizeBytes, &flags3, NULL, NULL);
WSARecv(video_socket, &sizebuf, 1, &sizeBytes, &flags3, NULL, NULL);
continue;}

first=FALSE;
```

```
if(strcmp("Finished",sizebuf.buf)==0)
{End=true;
}else
{size = atoi(sizebuf.buf);
summer = summer + size;

// Work out how many times to loop for this block of video data

recvBuffer.len = 240;                          // 240 bytes was derived from trial and error to be the
recvBuffer.buf = (char *) malloc(240);         // best value to use for breaking the large chunk of data
                                               // into smaller chunks. It approximates 5 ATM cells
loop = size/240;
if((size%240)  0)
loop++;

start_time = GetCurrentTime();                 // Now we can receive the next block of video data

for( y=0;y<loop;y++)
{
if (WSARecv(video_socket, &recvBuffer, 1, &recvBytes, &flags2, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error Receiving Video data : in GetVideo()");
char buffer[20];
_itoa( summer, buffer, 10 );
AfxMessageBox(buffer);
break;}

end_time = GetCurrentTime();
recvBuffer.len = recvBytes;
count = 0;

if( Buffer != NULL )
{if( !(Buffer-WriteBuffer( recvBuffer.buf, recvBytes )) )
AfxMessageBox("Not enough space in Buffer!!");
}else AfxMessageBox( "Buffer NULL" );
}if( (end_time) - (start_time) != 0)
{  data_rate = size * 8 / (end_time - start_time);
itoa(data_rate,rate,10);
strcat(rate,"\n");
videofile.Write(rate,10); };

free(sizebuf.buf);
free(recvBuffer.buf); } } }

void CATM::Close_Winsock()
{WSACleanup(); }                               // Close the socket and exit Winsock2.2 gracefully


/*******************************************************/
/* Function Name:      Broker_Connect                  */
/* Function:           To connect the Client to a Broker */
/* Socket:             broker_socket                   */
/*******************************************************/

BOOL CATM::Broker_Connect(SOCKET broker_socket, char *broker_name)

{

// Operation of Broker_Connect
//          1.Retrieve the ATM address of the broker from the ATM hosts file and
//          place it in the sock_addr data structure.
//          2.Provide the parameters for sock_addr
//          3.Connect to the broker - pass QOS requirements in the same call
//          4.Get the socket_ID, connection time, device_ID,vpi and vci values
//          5.Fill up the Network struct with all network details
//          This function uses the broker names in d:\winnt\atmhosts

DWORD bytes = 0;
struct sockaddr_atm sock_addr;
```

```
memset((void *)&sock_addr, 0, sizeof(struct sockaddr_atm));

if (get_atm_address( broker_name, &sock_addr) == FALSE)

{AfxMessageBox("Error : ATM NSAP Address not found for server");
return FALSE; }

sock_addr.satm_family          = AF_ATM;
sock_addr.satm_number.AddressType  = ATM_NSAP;
sock_addr.satm_number.NumofDigits  = ATM_ADDR_SIZE;
sock_addr.satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

char convert[2];
UCHAR a;
CString broker_address;
for(int p = 0; p < 20; p ++ )
{  a = sock_addr.satm_number.Addr[p];
sprintf(convert,"%02x",(char*)a);
broker_address += convert; };

// Attempt to connect to the broker with the broker_socket

if (WSAConnect(broker_socket, (struct sockaddr FAR *)&sock_addr,
sizeof(struct sockaddr_atm), NULL, NULL, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error : Broker Unavailable - please try to connect at a later stage.");
return FALSE; }

WSABUF id;                                          // Send a "C" to inform the broker that it is a client connection
id.len = 1;
id.buf = (char *) malloc(1);
ULONG b = 0 ;
ULONG f = 0;
id.buf = "C";
if((WSASend(broker_socket, &id,1, &b, f, NULL, NULL)) == SOCKET_ERROR)
{AfxMessageBox("Error sending C character - here's why...");
int y;
char * error;
_itoa(y,error,10);
AfxMessageBox(error); };

id.len = 20;                                        // Send this client's name to the broker
id.buf = (char *) malloc(20);
b = 0 ;
f = 0;
id.buf = client_name.GetBuffer(20);

if((WSASend(broker_socket, &id,1, &b, f, NULL, NULL)) == SOCKET_ERROR)
{AfxMessageBox("Error sending client name - here's why...");
int y;
char * error;
_itoa(y,error,10);
AfxMessageBox(error); };

char *sender = client_address.GetBuffer(40);        // Send the Client Address to the broker
WSABUF ClientAddress;
ULONG xmitBytes = 0;
ULONG flags = 0;
ClientAddress.len = 40;
ClientAddress.buf = (char *) malloc(40);
ClientAddress.buf = sender;
WSASend(broker_socket, &ClientAddress, 1, &xmitBytes, flags, NULL, NULL);

CTime time = CTime::GetCurrentTime();               // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

char sock_as_string[10];                            // Get the socket ID as string
_itoa(control_socket,sock_as_string,10);
```

```
ATM_CONNECTION_ID connID;                          // Get the VPI, VCI, DeviceNumber

if(WSAIoctl(broker_socket,SIO_GET_ATM_CONNECTION_ID,NULL,
0,(LPVOID)&connID,sizeof(ATM_CONNECTION_ID),&bytes,NULL,NULL) == SOCKET_ERROR)
AfxMessageBox("Error getting broker_socket connection ID.");

char devicenum[10];                                // Put the VPI, VCI, DeviceNumber in a string format
char vpi[10];
char vci[10];
int q = connID.DeviceNumber;
_itoa(q,devicenum,10);
q = connID.VPI;
_itoa(q,vpi,10);
q = connID.VCI;
_itoa(q,vci,10);

Network.client_name = client_name;                 // Connected to the broker, fill up the Struct with the details
Network.client_address = client_address;
Network.broker_name = broker_name;
Network.broker_address = broker_address;
Network.broker_time = connection_time;
Network.broker_socket_ID = sock_as_string;
Network.broker_VPI = vpi;
Network.broker_VCI = vci;
Network.broker_device_ID = devicenum;
return TRUE; }

/*****************************************************************/
/* Function Name:      get_atm_address                           */
/* Function:           Resolve the name string such as "server" into */
/*                     an actual ATM address                     */
/*****************************************************************/

BOOL CATM::get_atm_address(char *name, struct sockaddr_atm *atm_addr)
{DWORD dwValue;
HANDLE hLookup;
WSAQUERYSETW qsRestrictions;
CSADDR_INFO csaBuffer;
WCHAR tmpWStr[100];
MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, name, -1, tmpWStr, 100);
csaBuffer.LocalAddr.iSockaddrLength = sizeof (struct sockaddr_atm);
csaBuffer.LocalAddr.lpSockaddr = (struct sockaddr *) atm_addr;
csaBuffer.RemoteAddr.iSockaddrLength = sizeof (struct sockaddr_atm);
csaBuffer.RemoteAddr.lpSockaddr = (struct sockaddr *) atm_addr;
qsRestrictions.dwSize               = sizeof (WSAQUERYSETW);
qsRestrictions.lpszServiceInstanceName = NULL;
qsRestrictions.lpServiceClassId     = &FORE_NAME_CLASS;
qsRestrictions.lpVersion            = NULL;
qsRestrictions.lpszComment          = NULL;
qsRestrictions.dwNameSpace          = FORE_NAME_SPACE;
qsRestrictions.lpNSProviderId       = NULL;
qsRestrictions.lpszContext          = L"";
qsRestrictions.dwNumberOfProtocols  = 0;
qsRestrictions.lpafpProtocols       = NULL;
qsRestrictions.lpszQueryString      = tmpWStr;
qsRestrictions.dwNumberOfCsAddrs    = 1;
qsRestrictions.lpcsaBuffer          = &csaBuffer;
qsRestrictions.lpBlob               = NULL;

if (WSALookupServiceBeginW(&qsRestrictions, LUP_RETURN_ALL, &hLookup) == SOCKET_ERROR)
{AfxMessageBox("Socket error : WSALookupServiceBegin");
return FALSE;}
dwValue = sizeof (WSAQUERYSETW);
if (WSALookupServiceNextW ( hLookup, 0, &dwValue, &qsRestrictions) == SOCKET_ERROR)
{AfxMessageBox("Socket error : WSALookupServiceNext");
return FALSE;}

if (WSALookupServiceEnd (hLookup) == SOCKET_ERROR)
```

```
{AfxMessageBox("Socket error : WSALookupServiceEnd");
return FALSE;}

return TRUE; }

/*******************************************************/
/* Function Name:      Get_Videos                      */
/* Function:           Receive the video list from the broker  */
/* Socket:             broker_socket                   */
/*******************************************************/

void CATM::Get_Videos(BOOL FirstUse)
{
Send_Broker("G");                                   // Request the video list from the server

ULONG recvBytes = 0;                                // Now receive the movie list from the server
ULONG flags2 = 0;
WSABUF Videos;
Videos.len = 10000;
Videos.buf = (char *) malloc(10000);

if(WSARecv(broker_socket, &Videos, 1, &recvBytes, &flags2, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error Receiving List of Videos : in Get_Movies");};

videolist = (CString)Videos.buf;                    // Now parse the list into a temporary structure
vids_name.SetSize(100,1);
vids_category.SetSize(100,1);
vids_time.SetSize(100,1);
vids_description.SetSize(100,1);
int i=0;
int counter = 0;
char *ptr1,*ptr2,*ptr3,*ptr4;
ptr1 = mem_block;

while(videolist[i] != '\n')
{while(videolist[i]!='\r'){mem_block[i]=videolist[i];i++;};mem_block[i] = '\0';i++;ptr2=&mem_block[i];
while(videolist[i]!='\r'){mem_block[i]=videolist[i];i++;};mem_block[i] = '\0';i++;ptr3=&mem_block[i];
while(videolist[i]!='\r'){mem_block[i]=videolist[i];i++;};mem_block[i] = '\0';i++;ptr4=&mem_block[i];
while(videolist[i]!='\r'){mem_block[i]=videolist[i];i++;};mem_block[i] = '\0';i++;
vids_name.SetAt(counter,ptr1);
vids_time.SetAt(counter,ptr2);
vids_category.SetAt(counter,ptr3);
vids_description.SetAt(counter,ptr4);
counter++;
ptr1 = &mem_block[i]; }; }

/*******************************************************************/
/* Function Name:      Negotiate_QOS                               */
/* Function:           Send the user's movie choice to the broker. */
/*                     Use the broker_socket                       */
/*******************************************************************/

void CATM::Negotiate_QOS(CString name, int cancelled, int connected_to_server)

{  Network.video = name;
Send_Broker("Q");                                   // First send a "Q" to tell the broker we need to negotiate a QOS for this
                                                    // video
char *string;
if(cancelled==0)
{  string = name.GetBuffer(50);
Network.video = string;
}
else
string = "cancelled";
WSABUF moviesend;
ULONG xmitBytes = 0;
ULONG flags = 0;
moviesend.len = 50;
moviesend.buf = (char *) malloc(50);
```

```
moviesend.buf = string;
WSASend(broker_socket, &moviesend, 1, &xmitBytes, flags, NULL, NULL);

// Here is where the negotiation will take place....

// Now receive the name and ATM address of the server hosting this video

ULONG recvBytes = 0;
ULONG flags2 = 0;
WSABUF server_address;
server_address.len = 40;
server_address.buf = (char *) malloc(40);
if(WSARecv(broker_socket, &server_address, 1, &recvBytes, &flags2, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error Receiving server_address");};

CString s_address;                                  // Convert the server_address into the correct ATM form
s_address = server_address.buf;
s_address.GetBufferSetLength(40);
CString temp_copy = s_address;                      // Create an ATM-style char address for this server
char server_addr[20];
int index = 0;
int i=0;
int m=0;
char j;
char k;
while(i!=40)
{j = GetBinary(temp_copy[i]);
k =  GetBinary(temp_copy[i+1]);
j <<= 4;
server_addr[m] = j | k;
i += 2;
m++;};

CString server;                                     // Code for displaying an ATM address in "neat" format
UCHAR t;
char next_byte[2];
server.Empty();
for( int l = 0; l < 20; l ++ )
{  t = server_addr[l];
sprintf(next_byte,"%02x",(char*)t);
server += next_byte; };

// Connect to the server and off we go.We pass server_connect the server's name & atm address. f we are already connected to
// server then bypass  server connect else kill the current connection and connect to the new server

if(connected_to_server==0)
{Server_Connect(control_socket, server_addr, 0); }

if(connected_to_server==1)
{if(strcmp(s_address,Network.server_address)!=0)
{AfxMessageBox("Disconnecting...");
Server_Connect(control_socket, server_addr, 0); }; };

Send_Control("N");
WSASend(control_socket, &moviesend, 1, &xmitBytes, flags, NULL, NULL); }          // Send the video_name


/*****************************************************/
/* Function Name:      Server_Connect                */
/* Function:           To connect the Client to the Server  */
/*****************************************************/

BOOL CATM::Server_Connect(SOCKET control_socket, char *server_address, int sel)
{

// Operation of Server_Connect
//  1.  Place the server_address parameter in the sock_addr data structure
//  2.  Provide the parameters for sock_addr
//  3.  Connect to the server - pass QOS requirements in the same call
```

```
// 4. Send the client's ATM address to the server as a string of chars
// 5. Listen for the reverse connection from the server for the video channel
// 6. Get the socket_ID, connection time, device_ID,vpi and vci values
// 7. Fill up the Network struct with all network details.

ATM_CONNECTION_ID connID;
DWORD bytes;
bytes = 0;
struct sockaddr_atm sock_addr;
memset((void *)&sock_addr, 0, sizeof(struct sockaddr_atm));

// If it is required to enter the selector byte by hand then uncomment the next line:
// SockAddrAtm.satm_number.Addr[ATM_ADDR_SIZE-1] = (char) sel;

sock_addr.satm_family           = AF_ATM;
sock_addr.satm_number.AddressType  = ATM_NSAP;
sock_addr.satm_number.NumofDigits  = ATM_ADDR_SIZE;
sock_addr.satm_blli.Layer2Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;
sock_addr.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;

// Copy the server's address to sock_addr

memcpy(sock_addr.satm_number.Addr, server_address, ATM_ADDR_SIZE);

char next_byte[2];                                  // Code to print out an ATM address in neat format
UCHAR t;
CString client_addr;
for(int l = 0; l < 20; l ++ )
{  t = sock_addr.satm_number.Addr[l];
sprintf(next_byte,"%02x",(char*)t);
client_addr += next_byte; };
AfxMessageBox(client_addr);

// Attempt to connect to the server with the control_socket

if (WSAConnect(control_socket, (struct sockaddr FAR *)&sock_addr,
sizeof(struct sockaddr_atm), NULL, NULL, NULL, NULL) == SOCKET_ERROR)
{AfxMessageBox("Error connecting to server - please try to connect at a later stage.");
return FALSE; }

WSABUF name;                                        // Send this client's name to the broker
name.len = 20;
name.buf = (char *) malloc(20);
ULONG b = 0 ;
ULONG f = 0;
name.buf = client_name.GetBuffer(20);
if((WSASend(control_socket, &name,1, &b, f, NULL, NULL)) == SOCKET_ERROR)
{AfxMessageBox("Error sending client name - here's why...");
int y;
char * error;
_itoa(y,error,10);
AfxMessageBox(error); };

char *sender = client_address.GetBuffer(40);        // Send the client address
WSABUF ClientAddress;
ULONG xmitBytes = 0;
ULONG flags = 0;
ClientAddress.len = 40;
ClientAddress.buf = (char *) malloc(40);
ClientAddress.buf = sender;
WSASend(control_socket, &ClientAddress, 1, &xmitBytes, flags, NULL, NULL);

// Need to bind a socket to this client's address and then listen for the server to attempt to setup the video stream

struct sockaddr_atm binder;
char binder_addr[20];

for(l = -4; l < 16; l ++ )
```

```
{  binder_addr[l+4] = my_address.satm_number.Addr[l]; };      // Does conversion from UCHAR to char

binder_addr[19] = (char) 0x90;
memset((void*)&binder,0,sizeof(struct sockaddr_atm));
binder.satm_family          = AF_ATM;                        // Address Family is ATM
binder.satm_number.AddressType = ATM_NSAP;                   // Address Style is network service access point
binder.satm_number.NumofDigits = ATM_ADDR_SIZE;              // ATM_ADDR_SIZE = 20 (defined in ws2atm.h)
binder.satm_blli.Layer2Protocol = SAP_FIELD_ANY;             // Defined in ws2atm.h
binder.satm_blli.Layer3Protocol = SAP_FIELD_ABSENT;          // Defined in ws2atm.h
binder.satm_bhli.HighLayerInfoType = SAP_FIELD_ABSENT;       // Defined in ws2atm.h
memcpy(binder.satm_number.Addr,binder_addr,ATM_ADDR_SIZE);
binder.satm_number.Addr[19] = (char) 0x90;

if (bind(setup_video_socket, (struct sockaddr FAR *)&binder, sizeof(binder))== SOCKET_ERROR)
{char * error;
int err;
err = WSAGetLastError();
_itoa(err,error,10);
AfxMessageBox(error);
exit(0); }

if(listen(setup_video_socket, 5) == SOCKET_ERROR)
{AfxMessageBox("Error : listen ");
exit(0); }

struct sockaddr_atm ServerAddr;
int len;

if ((video_socket = WSAAccept(setup_video_socket,            // The socket which is listening for connections
(struct sockaddr FAR *)&ServerAddr,                          // Pointer to the buffer which receives the address of the
                                                             // connecting entity
&len,                                                        // The length of the address
NULL,
0))
== SOCKET_ERROR)
{
AfxMessageBox("Error : accept");
char error[10];
int err;
err = WSAGetLastError();
itoa(err,error,10);
AfxMessageBox(error); };

sock_addr.satm_number.Addr[19] = 0x0;
CString server_name;
server_name = Get_Name_By_Address(sock_addr,0);
sock_addr.satm_number.Addr[19] = 0x80;

CString server_addr;                                         // Get the server's address in a CString format

for(l = 0; l < 20; l ++ )
{t = sock_addr.satm_number.Addr[l];
sprintf(next_byte,"%02x",(char*)t);
server_addr += next_byte; };

char sock_as_string[10];                                     // Get the socket ID as string
_itoa(control_socket,sock_as_string,10);

CTime time = CTime::GetCurrentTime();                        // Get the connection time
CString connection_time = time.Format("%H:%M:%S  %A, %B %d, %Y");

// Get the VPI, VCI, DeviceNumber

if(WSAIoctl(control_socket,SIO_GET_ATM_CONNECTION_ID,NULL,
0,(LPVOID)&connID,sizeof(ATM_CONNECTION_ID),&bytes,NULL,NULL) == SOCKET_ERROR)
AfxMessageBox("error getting my connection ID.");

char devicenum[10];                                          // Put the VPI, VCI, DeviceNumber in a string format
char vpi[10];
```

```
char vci[10];
int q = connID.DeviceNumber;
_itoa(q,devicenum,10);
q = connID.VPI;
_itoa(q,vpi,10);
q = connID.VCI;
_itoa(q,vci,10);

Network.server_name = server_name;                    // Connected to the server, fill up the Struct with the details
Network.server_address = server_addr;
Network.client_address = client_address;
Network.server_socket_ID = sock_as_string;
Network.server_VPI = vpi;
Network.server_VCI = vci;
Network.server_device_ID = devicenum;
Network.server_time = connection_time;
return TRUE; }

/*****************************************************************/
/* Function Name:      Send_Control(char* type)                   */
/* Function:           Send a control signal to the server where control */
/*                     signal is P -play,C -cancel,S -stop       */
/*                     or N - new movie.                         */
/*                     Use the control_socket                    */
/*****************************************************************/

void CATM::Send_Control(char* type)
{WSABUF control;
ULONG xmitBytes = 0;
ULONG flags = 0;
control.len = 1;
control.buf = (char *) malloc(1);
control.buf = type;
WSASend(control_socket, &control, 1, &xmitBytes, flags, NULL, NULL);}

/*****************************************************************/
/* Function Name:      Send_Broker(char* type)                   */
/* Function:           Send a control signal to the broker where control */
/*                     signal is P -play,C -cancel,S -stop       */
/*                     or N - new movie.                         */
/*                     Use the control_socket                    */
/*****************************************************************/

void CATM::Send_Broker(char* type)
{WSABUF control;
ULONG xmitBytes = 0;
ULONG flags = 0;
control.len = 1;
control.buf = (char *) malloc(1);
control.buf = type;
WSASend(broker_socket, &control, 1, &xmitBytes, flags, NULL, NULL);}


void CATM::SetBuffer(CBuffer * Buf)
{Buffer = Buf; }

char CATM::GetBinary(char the_char)
{char the_return;
switch(the_char)
{case '0':  the_return = 0x0;break;
case '1':  the_return = 0x1;break;
case '2':  the_return = 0x2;break;
case '3':  the_return = 0x3;break;
case '4':  the_return = 0x4;break;
case '5':  the_return = 0x5;break;
case '6':  the_return = 0x6;break;
case '7':  the_return = 0x7;break;
case '8':  the_return = 0x8;break;
case '9':  the_return = 0x9;break;
```

```
case 'a':  the_return = 0xa;break;
case 'b':  the_return = 0xb;break;
case 'c':  the_return = 0xc;break;
case 'd':  the_return = 0xd;break;
case 'e':  the_return = 0xe;break;
case 'f':  the_return = 0xf;break;
default: AfxMessageBox("Error in GetBinary()'s case statement"); }
return the_return; }

CString CATM::Get_Name_By_Address(struct sockaddr_atm address, int offset)
{char* the_name;
CString name;
CString error = "Error";
int i;
atmhost_nb = 0;
char *filename = "D:\\winnt\\atmhosts";
Read_File(filename);
for (i=0;i<atmhost_nb;i++)
{atmhost_list[i].addr[19] = (char) 0x0;

if (memcmp(address.satm_number.Addr - offset,atmhost_list[i].addr,20)==0 && strcmp("localhost",atmhost_list[i].name)!=0)
{ the_name = atmhost_list[i].name;
name = CString(the_name);
return name; }; };
return error; }

void CATM::Parse_Line(char *buffer)
{ int i,j,n,k;
unsigned char addr[20];
char *start, *end;

// i: index into the parsed line j: inder into addr k: number of figure currently decoded remove trailing \r and \n

start = strrchr(buffer,'\r');
if (start != NULL) *start = 0;
start = strrchr(buffer,'\n');
if (start != NULL) *start = 0;

if (*buffer == 0)                                   // Skip empty lines
return;
memset(&addr,0,sizeof(addr));
n = strlen(buffer);
j = k = 0;
for (i=0;i<n;i++)
{ if ((buffer[i]='0'&&buffer[i]<='9')
|| (buffer[i]='a'&&buffer[i]<='f')
|| (buffer[i]='A'&&buffer[i]<='F'))
{if (k=2)                                           // We already read two figures, so read the next atm addr byte
{j++;
k = 0; }
if (j=20)
{return; }
addr[j] = addr[j]*16 + Get_Hex(buffer[i]);
k++;
}else if (buffer[i]=='.'||buffer[i]=='-')
{k = 0;
j++;
if (j=20)
{return; } }
else
{if (j==20 || (j==19 && k0))
break;
return; } }
start = buffer+i;                                   // addr is now correctly filled in, need to get the name
while ((*start==' '||*start=='\t') && *start!=0)
start++;
if (*start == 0)
{return; }
```

```
end = start;
while (*end!=' '&&*end!='\t'&&*end!='\r'&&*end!='\n'&&*end!=0)
end++;
*end = 0;
atmhost_list[atmhost_nb].name = strdup(start);
memcpy(atmhost_list[atmhost_nb].addr,addr,20);
atmhost_nb ++; }

void CATM::Read_File(const char* name)
{FILE *fp;
char buffer[1024];
fp = fopen(name,"r");
if (fp == NULL)
{ return; }

while (fgets(buffer,sizeof(buffer),fp)!= NULL)
{if (buffer[0] == ';' || buffer[0]=='#')
continue;
Parse_Line(buffer); }
fclose(fp); }

int CATM::Get_Hex(char c)
{if (c='0' && c<='9')
return (c-'0');
if (c='a' && c<='f')
return (c-'a'+10);
if (c='A' && c<='F')
return (c-'A'+10);
printf("internal error in name.c\n");
return 0; }
```

3.2 ATM.h

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <ws2atm.h>
#include "Buffer.h"
#include "afxmt.h"

class CATM : public CFrameWnd
{protected:
public:
CATM();
virtual ~CATM();
public:
int Init_Winsock(CString broker_name);
void GetVideo();
void SetBuffer( CBuffer * Buf );
void Negotiate_QOS(CString name,int cancelled,int connected_to_server);
void Close_Winsock();
void Get_Videos(BOOL FirstUse);
void Send_Control(char* type);
void Send_Broker(char* type);
char GetBinary(char the_char);
int Get_Hex(char c);
CString Get_Name_By_Address(struct sockaddr_atm address, int offset);
void Parse_Line(char *buffer);
void Read_File(const char* name);
CEvent *video_end;

struct NetworkDetails
{CString  client_name;
CString  client_address;
CString broker_name;
CString broker_address;
CString server_name;
```

```
CString server_address;
CString broker_time;
CString server_time;
CString broker_socket_ID;
CString broker_VPI;
CString broker_VCI;
CString broker_device_ID;
CString server_socket_ID;
CString server_VPI;
CString server_VCI;
CString server_device_ID;
CString video;
} Network;

struct atmhost
{char *name;
unsigned char addr[20];
} atmhost_list[20];

int atmhost_nb;
protected:
private:
CBuffer * Buffer;
BOOL Stop;
WSABUF recvBuffer;
WSABUF rec[20];
WSABUF xmitBuffer;
BOOL get_atm_address( char *, struct sockaddr_atm * );
BOOL Server_Connect( SOCKET, char*, int );
BOOL Broker_Connect( SOCKET, char*);
SOCKET setup_video_socket;
SOCKET video_socket;
SOCKET control_socket;
SOCKET broker_socket;
struct sockaddr_atm my_address;
CString client_name;
CString client_address; };
```

3.3 Buffer.cpp

```
#include "Buffer.h"
#include "stdafx.h"
#include <afxmt.h>                                    // Needed for CCriticalSection class.
HANDLE m_hOutputFile;
void CloseFile();
BOOL OpenFile();
void WriteFrame( UINT size );

CCriticalSection * Crit_Section;

// Must be global due to Buffer.h being included into both DxShow files and MFC files - if this is a member of Buffer.h get error
// regarding windows.h included - can't be with MFC, but must be with DxShow

CBuffer::CBuffer()
{StartBuffer = NULL;
BufferSize = 0;
HeadPos = 0;
TailPos = 0;
First = 1;
Crit_Section = new CCriticalSection();
LookBackBuffer[0] = LookBackBuffer[1] = LookBackBuffer[2] = LookBackBuffer[3] = 0xff;
OpenFile(); }

CBuffer::~CBuffer()
{RemoveBuffer();
CloseFile();
delete Crit_Section; }
```

```
// The read buffer method that the decoder calls to read from this buffer

int CBuffer::ReadBuffer( unsigned char * bufPTR, int BytesToRead )
{int count = 0;
int loop = 1;
unsigned char * CopyPTR = bufPTR;
if( BytesToRead == 0 )
return 0;
Crit_Section-Lock();                                    // Lock the buffer's critical section

while(First)
{if( SizeOfBuffer() = BytesToRead )
First = 0;
else
{Crit_Section-Unlock();
Sleep(33);
Crit_Section-Lock();                                    // Lock the buffer's critical section
} }

if( (BytesToRead - count)  SizeOfBuffer())
{ while( TailPos != HeadPos )                           // Must check if it is not end of file - to drop out.
{*CopyPTR++ = *(StartBuffer + TailPos);                 // Copy from the buffer to the decoder array
TailPos++;
count++;

if( TailPos == BufferSize )                             // Wrap the buffer around when it comes to the end
TailPos = 0;
}
if( TailPos == HeadPos )                                // Case of end of file reached or no Packet Start code in range to
                                                        // use to break and stuff
{while( count < BytesToRead )
{while( TailPos == HeadPos )
{Crit_Section-Unlock();                                 // Unlock the buffer's critical section
Sleep(10);
Crit_Section-Lock();                                    // Lock the buffer's critical section
if( TailPos == HeadPos )
{switch (count)                                         // Empty state - must check to see if end of file is reached
{case 0:  break;
case 1:  if (*(CopyPTR-1) == 0xB9)
{return count;}
else
break;
case 2:  if ((*(CopyPTR-1) == 0xB9) && (*(CopyPTR-2) == 0x01 ))
{return count;}
else
break;
case 3:  if ((*(CopyPTR-1) == 0xB9) && (*(CopyPTR-2) == 0x01 ) &&
(*(CopyPTR-3) == 0x00))
{return count; }
else
break;
default: if ((*(CopyPTR-1) == 0xB9) && (*(CopyPTR-2) == 0x01 ) &&
(*(CopyPTR-3) == 0x00) && (*(CopyPTR-4) == 0x00 ))
{return count; }}}}

*CopyPTR++ = *(StartBuffer + TailPos);                  // Copy from the buffer to the decoder array
TailPos++;
count++;

if( TailPos == BufferSize )                             // Wrap the buffer around when it comes to the end
TailPos = 0; } }
else
{*CopyPTR++ = 0x00;
*CopyPTR++ = 0x00;
*CopyPTR++ = 0x01;
*CopyPTR++ = 0xBE;

int Packet = BytesToRead - count - 6;
if(( Packet < 0 ) || ( Packet  BytesToRead ))
```

```
{Packet = 1;}

*CopyPTR++ = Packet  8;
*CopyPTR++ = Packet;
*CopyPTR++ = 0x0F;

count += 7;
while( Packet  1 )
{*CopyPTR++ = 0xFF;
Packet--;
count++; }
AfxMessageBox("Stuffing Packets"); } }
else
{while( count < BytesToRead )
{*CopyPTR++ = *(StartBuffer + TailPos);          // Copy from the buffer to the decoder array
TailPos++;
count++;

if( TailPos = BufferSize )                       // Wrap the buffer around when it comes to the end.
TailPos = 0; } }
Crit_Section-Unlock();                           // Bytes copies, unlock the critical section and return number of
                                                 // bytes copied
return count; }

// Method to set up the buffer to the correct size for the incoming data
char CBuffer::SetUpBuffer(int MFmax)
{// Allocate the needed memory - amount needed = MFmax parameter - size of maxiumum macroframe
BufferSize = MFmax;
if((StartBuffer = new char[BufferSize]) == NULL)
{AfxMessageBox("StartBuffer not allocated!  Memory Error!");
return FALSE; }
TailPos = 0;
HeadPos = 0;                                     // Empty condition
return TRUE; }

// Removes the buffer that is set up in the SetUpBuffer method
void CBuffer::RemoveBuffer()
{if(StartBuffer != NULL)                         // If the buffer has been created
{delete [] StartBuffer;                          // Delete the allocated memory and reset the variables
HeadPos = 0;
TailPos = 0;
BufferSize = 0;
StartBuffer = NULL; } }

void CBuffer::ResetBuffer( int difference )
{int TempTail;
Crit_Section-Lock();
TempTail = TailPos - difference;
if( TempTail < 0 )
TempTail = BufferSize + TempTail;
if( TempTail  BufferSize )
TempTail = TempTail % BufferSize;
if( TailPos < HeadPos )
if(( TempTail < HeadPos ) && ( TempTail  TailPos ))
printf( "Error in Reseting!!!" );
else
TailPos = TempTail;
else
if(( TempTail  HeadPos ) && ( TempTail < TailPos ))
printf( " Error in Reseting!!!" );
else
TailPos = TempTail;
Crit_Section-Unlock(); }

char CBuffer::WriteBuffer( char * wMem, unsigned long wSize )
{unsigned int count = 0;
ULONG bWrite;
Crit_Section-Lock();
```

```
while((( HeadPos != TailPos-1 ) &&
!(( HeadPos == BufferSize ) && ( TailPos == 0 ))) &&
(count < wSize))
{if( HeadPos = BufferSize )
HeadPos = 0;
LookBackBuffer[0] = LookBackBuffer[1];
LookBackBuffer[1] = LookBackBuffer[2];
LookBackBuffer[2] = LookBackBuffer[3];
LookBackBuffer[3] = *(StartBuffer + HeadPos) = *(wMem + count);

if(( LookBackBuffer[0] == 0x00 ) && ( LookBackBuffer[1] == 0x00 ) &&
( LookBackBuffer[2] == 0x01 ) && ( LookBackBuffer[3] == 0xba ))
{PacketStartPosition = HeadPos - 3;
if(( PacketStartPosition < 0 ) || ( PacketStartPosition  BufferSize )) // Accounts for wrap around on a UINT
PacketStartPosition += BufferSize; }

HeadPos++;
count++;}

Crit_Section-Unlock();

if(count != wSize)
{AfxMessageBox("Buffer full!!");
return 0;}
else
return 1;}

int CBuffer::SizeOfBuffer()
{if( TailPos <= HeadPos )
return( HeadPos - TailPos );
else
return( BufferSize - TailPos + HeadPos ); }

BOOL OpenFile()                                          // Open the respective files

{m_hOutputFile = CreateFile( "buffer.txt", GENERIC_WRITE,
FILE_SHARE_READ, NULL,
CREATE_ALWAYS, 0, NULL );
if (m_hOutputFile == INVALID_HANDLE_VALUE)
{AfxMessageBox("Could not open Output File");
return FALSE;}
return TRUE;}

void CloseFile()                                         // Close the respective files when finished

{CloseHandle( m_hOutputFile );
m_hOutputFile = INVALID_HANDLE_VALUE;}

void CBuffer::ResetBuffer()
{TailPos = 0;}
```

3.4 Buffer.h

```
class CBuffer
{public:
CBuffer();
public:
void ResetBuffer();
char WriteBuffer( char * wMem, unsigned long wSize );
void ResetBuffer( int difference );
void RemoveBuffer();                                     // Removes a buffer's memory that has been allocated
char SetUpBuffer(int MFmax);                             // Allocates space and sets up a new buffer
int ReadBuffer( unsigned char * bufPTR, int dwBytesToRead ); // Read from the buffer
int SizeOfBuffer();
virtual ~CBuffer();
private:
char First;
unsigned int PacketStartPosition;
```

```
char LookBackBuffer[4];
unsigned int  HeadPos;                          // Head position pointer
unsigned int  TailPos;                          // Tail position pointer
unsigned int  BufferSize;                       // End of buffer pointer
char * StartBuffer; };                          // Start of buffer pointer
```

3.5 Decoder.cpp

```
#include "Decoder.h"
#include <stdio.h>
#define WM_GRAPHNOTIFY  WM_USER+13
#define HELPER_RELEASE(x) { if (x) x-Release(); x = NULL; }

CDecoder::CDecoder()
{PlayWin = NULL;
Playing = FALSE;
pifPP = NULL;
pispp = NULL;}

CDecoder::~CDecoder() {}

BOOL CDecoder::Decode(CBuffer * Buffer, void * pPlayWin)
{PlayWin = (HWND)pPlayWin;
CMediaType mt;
mt.majortype = MEDIATYPE_Stream;
mt.subtype = MEDIASUBTYPE_MPEG1System;          //VideoCD;//MPEG1System//Video
mt.SetVariableSize();
HRESULT hr = S_OK;
CoInitialize(NULL);
CMemStream Stream(Buffer);
CMemReader *rdr = new CMemReader(&Stream, &mt, &hr);
if (FAILED(hr) || rdr == NULL) {
delete rdr;
printf("Could not create filter HRESULT ");
CoUninitialize();
return 1;}

rdr-AddRef();
pFG = NULL;
hr = SelectAndRender(rdr, &pFG);
if (FAILED(hr)) {
printf( "Failed to create graph and render file HRESULT ");
return 1;
} else {
HRESULT hr = PlayFileWait(pFG);                 // Play the file
if (FAILED(hr)) {
printf("Failed to play graph HRESULT ");
return 1; } }
rdr-Release();
if (pFG) {
ULONG ulRelease = pFG-Release();
if (ulRelease != 0) {
printf("Filter graph count not 0!  was ..");
return 1; } }
CoUninitialize();
return 0; }

HRESULT CDecoder::SelectAndRender(CMemReader * pReader, IFilterGraph * * ppFG)
{
HRESULT hr = CoCreateInstance(CLSID_FilterGraph,    // Create filter graph
NULL,
CLSCTX_INPROC,
IID_IFilterGraph,
(void**) ppFG);

if (FAILED(hr)) {
return hr;
}hr = (*ppFG)-AddFilter(pReader, NULL);          // Add our filter
```

```
if (FAILED(hr)) {
return hr;}
IGraphBuilder *pBuilder;                                    // Render our output pin
hr = (*ppFG)-QueryInterface(IID_IGraphBuilder, (void **)&pBuilder);
if (FAILED(hr)) {
return hr;}
hr = pBuilder-Render(pReader-GetPin(0));
pBuilder-Release();
return hr;}

HRESULT CDecoder::PlayFileWait(IFilterGraph * pFG)
{RECT grc;
pVW = NULL;
HRESULT hr = pFG-QueryInterface(IID_IMediaControl, (void **)&pMC);
if (FAILED(hr)) {
return hr;}

hr = pFG-QueryInterface(IID_IMediaEventEx, (void **)&pME);
if (FAILED(hr)) {
pMC-Release();
return hr;}

hr = pFG-QueryInterface(IID_IVideoWindow, (void **)&pVW);
if (FAILED(hr)) {
pMC-Release();
pFG-Release();
return hr;}

pVW-put_Owner((OAHWND)PlayWin);
pVW-put_WindowStyle(WS_CHILD|WS_CLIPSIBLINGS|WS_CLIPCHILDREN);

// Have the graph signal event via window callbacks for performance
pME-SetNotifyWindow((OAHWND)PlayWin, WM_GRAPHNOTIFY, 0);
GetClientRect(PlayWin, &grc);
pVW-SetWindowPosition(grc.left, grc.top, grc.right, grc.bottom);
OAEVENT oEvent;
hr = pME-GetEventHandle(&oEvent);
if (SUCCEEDED(hr)) {
hr = pMC-Run();}
if (SUCCEEDED(hr)) {
Playing = TRUE;
LONG levCode;
hr = pME-WaitForCompletion(INFINITE, &levCode);
Playing = FALSE;}

GetPropertyPage();
pVW-put_Owner(NULL);
pMC-Release();
pME-Release();
pVW-Release();
return hr; }

void CDecoder::ResetWin()
{if(Playing)
{RECT grc;
GetClientRect(PlayWin, &grc);
pVW-SetWindowPosition(grc.left, grc.top, grc.right, grc.bottom); } }

void CDecoder::GetPropertyPage()
{
// The easiest way to find the single MPEG video codec in the graph is via IBaseFilter::FindFilterByName. Multiple instances of
// the same filter make things a bit more challenging...Video Renderer
pFG-FindFilterByName(L"Video Renderer", &pifPP);
pifPP-QueryInterface(IID_ISpecifyPropertyPages, (void **)&pispp);

pispp-GetPages(&caGUID);                                    // Declare the counted array of GUIDs for the property page

HELPER_RELEASE(pispp);
```

```
OleCreatePropertyFrame(NULL,                    // Display default MPEG Video Decoder filter's property page
0,
0,
L"Filter",
1,
(IUnknown **)&pifPP,
0,
NULL,
0,
0,
NULL);
HELPER_RELEASE(pifPP); }

void CDecoder::SetRun()
{HRESULT hr;
hr = pMC-Run();}

void CDecoder::SetPaused()
{HRESULT hr;
hr = pMC-Pause();}
```

3.6 Decoder.h

```
#include "buffer.h"
#include <streams.h>
#include "memrdr.h"

class CDecoder
{public:
void SetPaused();
void SetRun();
void GetPropertyPage();
BOOL Decode( CBuffer * Buffer, LPVOID pPlayWin);
void ResetWin();
CDecoder();
virtual ~CDecoder();
private:
BOOL Playing;
CAUUID caGUID;
ISpecifyPropertyPages * pispp;
IBaseFilter * pifPP;
IMediaEventEx * pME;
IMediaControl * pMC;
IFilterGraph * pFG;
IVideoWindow * pVW;
HWND PlayWin;
HRESULT PlayFileWait( IFilterGraph *pFG );
HRESULT SelectAndRender( CMemReader *pReader, IFilterGraph **ppFG );};
```

4. ToolBox Classes

4.1 FrameReader.cpp

```
#include "stdafx.h"
#include "Toolbox.h"
#include "FrameReader.h"
#include "DiskReader.h"
#define MPEG_BUFFERSIZE 131072
UINT size;

FrameReader::FrameReader()
{ // Below are two checks to ensure if frame reading occurs before file opened - it will return nothing.
FileFinished = TRUE;
FillNextBuffer = FALSE;
counter = 0;
```

```
Locked = FALSE;
Frames = 0;
size = 0;
position = 0;
BP_position = 0;
I_Frame_Total = 0;
B_Frame_Total = 0;
P_Frame_Total = 0;
current_type = 3;                                   // i.e. not I=0, B=1 or P=2
GOP_I_Frame_Total = 0;
GOP_B_Frame_Total = 0;
GOP_P_Frame_Total = 0;
bp_ratio = 0;
bp_calc_first=1; }

FrameReader::~FrameReader()
{if( MPEGFileBuffer != NULL )
delete [] MPEGFileBuffer;}

void FrameReader::OpenFile()
{ MPEGFileBufferPosition = 0;
MPEGFileBufferEnd = MPEG_BUFFERSIZE;
First = TRUE;
MPEGFileBuffer = new BYTE[MPEG_BUFFERSIZE];
if( MPEGFileBuffer == NULL )
AfxMessageBox("CFrameReader::OpenMPEGFile() - Not enough memory to allocate 2K for MPEGFileBuffer" );
disk = new DiskReader( MPEGFileBuffer, this );
AfxBeginThread( StartDisk, disk, THREAD_PRIORITY_TIME_CRITICAL );
FileFinished = FALSE;
disk-OpenFile("d://MPEG Files//porsche.mpg");
disk-FillTotalBuffer();                             // Must fill the buffer initially

// Load Initial values into Look Ahead Buffer. Fill up the Lookahead buffers  Should be 00 00 01 BA 33 for example

LookAheadBuffer[0] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[1] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[2] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[3] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[4] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++]; }

// This function starts the while loop in the ReadAhead function in DiskReader. The while loop waits for a Readevent. A
// ReadEvent is signalled by a DiskReader.DoRead()

UINT FrameReader::StartDisk( LPVOID pParam )
{  DiskReader * TheDisk;
TheDisk = (DiskReader *) pParam;
TheDisk-ReadAhead();
return 0; }

void FrameReader::CountFrames()

{ DWORD size;
DWORD FramesRead = 0;
BOOL last = FALSE;
ULONG bWrite;
int TotalSize = 0;
BYTE * buf;
buf = new BYTE[1000000];

while( !last && FramesRead <= 67000 )               // Allows for 37 minutes of video (67000 frames at 30fps)
{ size = GetMacroFrame( buf, &last );
if( size  0 )
{  FramesRead++;
TotalSize += size;
SetData( size );     }  }

char * I_Total = new char[10];
itoa( I_Frame_Total, I_Total, 10 );
```

```
CString I_Tot = (CString) I_Total;
AfxMessageBox(I_Tot);

char * B_Total = new char[10];
itoa( B_Frame_Total, B_Total, 10 );
CString B_Tot = (CString) B_Total;
AfxMessageBox(B_Tot);

char * P_Total = new char[10];
itoa( P_Frame_Total, P_Total, 10 );
CString P_Tot = (CString) P_Total;
AfxMessageBox(P_Tot);

char * num_frames = new char[10];                    // Log to file
int frms = (int) FramesRead;
itoa( frms, num_frames, 10 );
CString number_of_frames = (CString) num_frames;

char * sz = new char[10];
itoa( TotalSize, sz, 10 );
CString sze = (CString) sz;
int total_length;

CString precursor1 = "Toolbox Version 0.1 Copyright Matthew Roux 1999.\nFunction: Frame Sizes\nVideoname: ";
CString filename = "d://MPEG Files//frame sizes.txt";
CString videoname = "around.mpg";
CString postcursor1 = "\nTotal Number of Frames: ";
CString postcursor2 = "\nTotal Size (in Bytes): ";
CString postcursor3 = "\n\n";

total_length = precursor1.GetLength() +  videoname.GetLength() +
postcursor1.GetLength() + number_of_frames.GetLength() + postcursor2.GetLength() +
+ sze.GetLength() + postcursor3.GetLength();

CString name = precursor1  + videoname + postcursor1 + number_of_frames
+ postcursor2 + sze + postcursor3;

m_hOutputFile = CreateFile( filename, GENERIC_WRITE,
FILE_SHARE_READ, NULL,
CREATE_ALWAYS, 0, NULL );
if (m_hOutputFile == INVALID_HANDLE_VALUE)
{AfxMessageBox("Could not open Output File"); exit(0);}

WriteFile( m_hOutputFile, name, total_length, &bWrite, NULL );

SaveData();
CloseHandle( m_hOutputFile );
m_hOutputFile = INVALID_HANDLE_VALUE;

CString bp_filename = "d://MPEG Files//bp file.txt";    // Save the BP data to a file
BP_OutputFile = CreateFile( bp_filename, GENERIC_WRITE,
FILE_SHARE_READ, NULL,
CREATE_ALWAYS, 0, NULL );
if (BP_OutputFile == INVALID_HANDLE_VALUE)
{AfxMessageBox("Could not open Output File"); exit(0);}
BP_SaveData();
CloseHandle( BP_OutputFile );
BP_OutputFile = INVALID_HANDLE_VALUE;
last = FALSE;
FramesRead = 0; }


// Manipulate the bit level file to get a macroframe.

UINT FrameReader::GetMacroFrame(BYTE * mpeg_buffer, BOOL * LastBuffer )
{ UINT VideoSize = 1;
UINT VSumSize = 0;                                   // This is the individual size of each frame. When amount is 1,
                                                     // this function is called n times where n is the number of frames
                                                     // in the video
```

```
int NumFrames = 0;
int numframes = 1;
int * amount = &numframes;
do
{VideoSize = GetFrame( (mpeg_buffer+VSumSize));

// Here we calculate the B:P ratio for each GOP

if(current_type == 0)                                   // i.e. an I frame / start of GOP

{                                                       // Calculate the BP ratio for the current GOP
if(bp_calc_first) bp_calc_first = 0;

else
{ if( (GOP_B_Frame_Total != 0) && (GOP_P_Frame_Total != 0) )
bp_ratio = (GOP_B_Frame_Total / GOP_P_Frame_Total) * 4/10;

BP_SetData(bp_ratio);                                   // Put it in the data array for later saving

GOP_B_Frame_Total = 0;                                  // Clear the current GOP stats for B and P frames
GOP_P_Frame_Total = 0;          }       }

if(current_type == 1)
{  GOP_B_Frame_Total =  GOP_B_Frame_Total + VideoSize; }

if(current_type == 2)
{  GOP_P_Frame_Total = GOP_P_Frame_Total +  VideoSize;  }

counter += VideoSize;
VSumSize += VideoSize;

if(( VideoSize  0 ) &&
((LookAheadBuffer[5] & 0x18) != 0x00))
{
NumFrames++;
Frames++;                                               // A running total of the number of frames
size += VSumSize;  }                                    // size is a running total of the whole video size

if( Frames == 67000 )
{
char sizeSTR[20];
itoa( size, sizeSTR, 10 );
AfxMessageBox("More than 67000 frames");
AfxMessageBox( sizeSTR );                               // i.e. too many frames
} }

while ( ( (*amount)  NumFrames ) &&
( VideoSize  0 ) &&
(*(mpeg_buffer+VSumSize-1) != 0xB9 ));

if(( NumFrames == 0 ) && ( VSumSize  0 ) &&
( FileFinished ) &&
( MPEGFileBufferPosition == MPEGFileBufferEnd ))
*amount = 1;
else
*amount = NumFrames;

if(( ( *amount == 1 ) || ( VideoSize == 0 )) && ( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{  // Set that last buffer of video has been filled and sets file back to beginning
counter = 0;
*LastBuffer = TRUE;
ResetMPEGFile();
}  return VSumSize;  }                                  // Return the size of this frame

// Manipulates the bit level file to get a frame. In all cases except the first time entering the procedure, the LookAheadBuffer
// will be on a 0x00 00 01 border, thus this must be accounted for

UINT FrameReader::GetFrame( BYTE *pFrame )
{ int type;
```

```
BYTE * FramePTR = pFrame;
UINT FrameSize = 0;
if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{return 0;}

while(!(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd )))
{
// We need to check what sort of frame we are busy counting so that we can label and do a running total of I,P and B sizes

if  (  (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00) &&
(LookAheadBuffer[4]== 0x00) && ((LookAheadBuffer[5] & 0x18) == 0x08) )

{  type = 0;                                                    // We have an I_Frame
current_type = 0;  }

if (  (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00) &&
((LookAheadBuffer[5] & 0x18) == 0x18) )

{  type = 1;                                                    // We have a B_Frame
current_type = 1;  }

if (  (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00)  &&
(LookAheadBuffer[4] != 0x00) && ((LookAheadBuffer[5] & 0x18) == 0x10))

{  type = 2;                                                    // We have a P_Frame
current_type = 2;  }

if(First)
First = FALSE;
else
FramePTR = MoveOffCurrentStartCode( FramePTR, &FrameSize );
FramePTR = FindNextStartCode( FramePTR, &FrameSize, 999999 );

if(( FileFinished ) && ( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{ if( LookAheadBuffer[3] != 0xB9 )
{ *FramePTR++ = 0x00;
*FramePTR++ = 0x00;
*FramePTR++ = 0x01;
*FramePTR++ = 0xB9;
FrameSize += 4;
return FrameSize;  }  }

if( LookAheadBuffer[3] == 0x00 )                                // Frame beginning
{  switch(type) {
case 0: I_Frame_Total += FrameSize;break;
case 1: B_Frame_Total += FrameSize;break;
case 2: P_Frame_Total += FrameSize;break;
default: break;
}  return FrameSize;  }
}   return FrameSize;  }

BYTE * FrameReader::MoveOffCurrentStartCode(BYTE * FramePTR, UINT * FrameSize )
{ *FramePTR++ = LookAheadBuffer[0];
*FrameSize += 1;

LookAheadBuffer[0] = LookAheadBuffer[1];
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
if( FileFinished )
{ *FramePTR++ = LookAheadBuffer[0];
*FramePTR++ = LookAheadBuffer[1];
```

```
*FramePTR++ = LookAheadBuffer[2];
*FramePTR++ = LookAheadBuffer[3];
*FrameSize += 4;
return FramePTR; }

if( Locked )
{ ConflictSection.Unlock();
Locked = FALSE;
Sleep(0); }

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{
if( Locked )
{ ConflictSection.Unlock();
Locked = FALSE;
Sleep(0); }
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0); }

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0;
return FramePTR; }

// MaxFetch used in conjunction with GetBlock to limit amount fetched to amount wanted - in other calls the function use
// 999999 to ensure SCode found before 999999 bytes 999999 = larger than 640*480*3 byte uncompressed image

BYTE * FrameReader::FindNextStartCode(BYTE * FramePTR, UINT * FrameSize, UINT MaxFetch )
{  while(!((LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 )))
{  if( MaxFetch == 0 )
return FramePTR;
*FramePTR++ = LookAheadBuffer[0];
*FrameSize += 1;                                    // The frame size incrementer
MaxFetch--;

LookAheadBuffer[0] = LookAheadBuffer[1];
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
if( FileFinished )
{  *FramePTR++ = LookAheadBuffer[0];
*FramePTR++ = LookAheadBuffer[1];
*FramePTR++ = LookAheadBuffer[2];
*FramePTR++ = LookAheadBuffer[3];
*FrameSize += 4;
return FramePTR; }

if( Locked )
{ ConflictSection.Unlock();
Locked = FALSE;
Sleep(0); }

if( MPEGFileBufferPosition == MPEGFileBufferEnd )
{ if( Locked )
{ ConflictSection.Unlock();
Locked = FALSE;
Sleep(0);  }
while( MPEGFileBufferPosition == MPEGFileBufferEnd )
Sleep(0);   }
```

```
if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ ConflictSection.Lock();
Locked = TRUE;
disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
MPEGFileBufferPosition = 0;
}  return FramePTR; }

void FrameReader::SaveData()
{ for( DWORD loop=0; loop < position; loop++ )
WriteData( DataArray[ loop ] );
position = 0;}

void FrameReader::BP_SaveData()
{ for( DWORD loop=0; loop < BP_position; loop++ )
BP_WriteData( BP_DataArray[ loop ] );
BP_position = 0; }

void FrameReader::SetData( DWORD data )
{ if( position < 250000 )
{ DataArray[ position ] = data;
position++; }
else AfxMessageBox( "Data output exceeding 250000 units - make another plan" );}

void FrameReader::BP_SetData( float data )
{ if( BP_position < 5000 )
{ BP_DataArray[ BP_position ] = data;
BP_position++; }
else  AfxMessageBox( "Data output exceeding 5000 units - make another plan" );}

void FrameReader::WriteData( DWORD size )                    // Write the data to a file
{ char *sizeBuf;
ULONG bWrite;
sizeBuf = new char[10];
itoa( size, sizeBuf, 10 );

if( size < 10 )
{ sizeBuf[1] = '\n';
size = 2; }
else if( size < 100 )
{ sizeBuf[2] = '\n';
size = 3; }
else if( size < 1000 )
{ sizeBuf[3] = '\n';
size = 4; }
else if( size < 10000 )
{ sizeBuf[4] = '\n';
size = 5;  }
else if( size < 100000 )
{ sizeBuf[5] = '\n';
size = 6;  }
else if( size < 1000000 )
{ sizeBuf[6] = '\n';
size = 7; }
else if( size < 10000000 )
{ sizeBuf[7] = '\n';
size = 8;  }
else if( size < 100000000 )
{ sizeBuf[8] = '\n';
size = 9; }
else
{ sizeBuf[9] = '\n';
size = 10; }

WriteFile( m_hOutputFile, sizeBuf, size, &bWrite, NULL );
if( bWrite != size )
```

```
AfxMessageBox("Error writing to Output File "); }

void FrameReader::BP_WriteData( float size )                    // Write the data to a file
{ ULONG bWrite;
char value[20];
_gcvt(size, 8, value );
WriteFile( BP_OutputFile, value, 7, &bWrite, NULL );
WriteFile( BP_OutputFile, "\n", 1, &bWrite, NULL ); }

void FrameReader::SetEndOfMPEGBuffer( DWORD Position )
{ if(!FileFinished) {
ConflictSection.Lock();
MPEGFileBufferEnd = Position;
ConflictSection.Unlock();} }

void FrameReader::SetFileFinished()
{  FileFinished = TRUE; }

void FrameReader::ResetMPEGFile()
{ First = TRUE;
disk-FillTotalBuffer();
FileFinished = FALSE;
MPEGFileBufferEnd = MPEG_BUFFERSIZE;
MPEGFileBufferPosition = 0;

LookAheadBuffer[0] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[1] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[2] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[3] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[4] = MPEGFileBuffer[MPEGFileBufferPosition++];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++]; }

// The algorithm: (1) move off the current start code (2) find the next start code, (3) Is it a I, P or B frame? (4) If I frame,extract it

int FrameReader::FindElement(int * element)
{ DWORD FramesRead = 0;
BOOL last = FALSE;
int TotalSize = 0;

int moving_byte_counter = 0;
int I_Counter = 0;
int I_Total_Size = 0;
int B_Total_Size = 0;
int P_Total_Size = 0;
CString number_of_bytes;

BYTE AND_Result;
int which_loop = 0;

if(*element == 1) { AND_Result = 0x08; which_loop = 0;};      // Find I Frames  i.e  xxx0 1xxx
if(*element == 2) { AND_Result = 0x18; which_loop = 1;};      // Find B Frames  i.e  xxx1 1xxx
if(*element == 3) { AND_Result = 0x10; which_loop = 2;};      // Find P Frames  i.e  xxx1 0xxx


// Notes on MPEG frames:
// An I-frame looks as follows:  00 00 01 00 00 yy where yy is a byte which has the 4th bit = 1 and the 5th bit = 0
// as in xxx0 1xxx
// An B-frame looks as follows:  00 00 01 00 zz yy where yy is a byte which has the 4th bit = 1 and the 5th bit = 1
// as in xxx1 1xxx. zz is not equal to zero
// An P-frame looks as follows:  00 00 01 00 zz yy where yy is a byte which  has the 4th bit = 0 and the 5th bit = 1
// as in xxx1 0xxx. zz is not equal to zero


if(which_loop==0)                                             // The I-frame loop
{ while(!( MPEGFileBufferPosition == MPEGFileBufferEnd ))
{ while(!  ( (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00)
&& (LookAheadBuffer[4]== 0x00) && ((LookAheadBuffer[5] & 0x18) == AND_Result) ))
{
```

```
LookAheadBuffer[0] = LookAheadBuffer[1];                    // Move one byte forward
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];
moving_byte_counter++;

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
{  MPEGFileBufferPosition = 0;}
if(FileFinished)
break;   }

if(FileFinished)
{  ResetMPEGFile();
return I_Counter;}
I_Counter++;
char * bytes = new char[10];
itoa( moving_byte_counter, bytes, 10 );
number_of_bytes = (CString) bytes;

LookAheadBuffer[0] = LookAheadBuffer[1];                    // Move off current I-frame start code
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];
moving_byte_counter++; }  }

if(which_loop==1)                                           // The B-frame loop

{ while(!( MPEGFileBufferPosition == MPEGFileBufferEnd ))

{ while(!  ( (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00)
&& ((LookAheadBuffer[5] & 0x18) == AND_Result) ))

{ LookAheadBuffer[0] = LookAheadBuffer[1];                  // Move one byte forward
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];

moving_byte_counter++;

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
{  MPEGFileBufferPosition = 0;}
if(FileFinished)
break; }

if(FileFinished)
{  ResetMPEGFile();
return I_Counter;}
I_Counter++;
char * bytes = new char[10];
itoa( moving_byte_counter, bytes, 10 );
CString number_of_bytes = (CString) bytes;

LookAheadBuffer[0] = LookAheadBuffer[1];                    // Move off current I-frame start code
LookAheadBuffer[1] = LookAheadBuffer[2];
```

```
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];      } }

if(which_loop==2)                                          // The P-frame loop

{while(!( MPEGFileBufferPosition == MPEGFileBufferEnd ))

{ while(! ( (LookAheadBuffer[0] == 0x00 ) && (LookAheadBuffer[1] == 0x00) &&
(LookAheadBuffer[2] == 0x01 ) && (LookAheadBuffer[3] == 0x00)
&& (LookAheadBuffer[4] != 0x00) && ((LookAheadBuffer[5] & 0x18) == AND_Result) ))

{ LookAheadBuffer[0] = LookAheadBuffer[1];                 // Move one byte forward
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];
moving_byte_counter++;

if(( MPEGFileBufferPosition == MPEG_BUFFERSIZE/2 ) ||
( MPEGFileBufferPosition == MPEG_BUFFERSIZE ))
{ disk-DoRead(); }

if( MPEGFileBufferPosition = MPEG_BUFFERSIZE )
{  MPEGFileBufferPosition = 0;}
if(FileFinished)
break;  }

if(FileFinished)
{  ResetMPEGFile();
return I_Counter;}
I_Counter++;
char * bytes = new char[10];
itoa( moving_byte_counter, bytes, 10 );
CString number_of_bytes = (CString) bytes;

LookAheadBuffer[0] = LookAheadBuffer[1];                   // Move off current I-frame start code
LookAheadBuffer[1] = LookAheadBuffer[2];
LookAheadBuffer[2] = LookAheadBuffer[3];
LookAheadBuffer[3] = LookAheadBuffer[4];
LookAheadBuffer[4] = LookAheadBuffer[5];
LookAheadBuffer[5] = MPEGFileBuffer[MPEGFileBufferPosition++];      }   }
return I_Counter; }

void FrameReader::Calculate_BP_Ratio()
{  HANDLE input_file;                                      // Open the File
char * filename  = "c://unzipped//Rose's data sets//soccer_2_tar";
input_file = CreateFile(  filename,
GENERIC_READ,
FILE_SHARE_READ,
NULL,
OPEN_EXISTING,
0,
NULL);
if (m_hFile == INVALID_HANDLE_VALUE)
AfxMessageBox("CDiskReader::OpenFile():- Could not open MPEG file!");

ULONG AmtRead;
int position = 0;
int GOP_position = 0;
float GOP_Array[12];
char frame_array[7];
int counter = 0;
BYTE * input;
BYTE * input2;
BYTE check_for_line_feed = 0x0a;
BYTE check_for_end_of_file = 0x00;
```

```
int big_count=0;
float BP_average = 0;

while(big_count < 2000)
{  while(GOP_position != 12)
{ do
{ ReadFile( input_file, input2, 1, &AmtRead, NULL );
frame_array[counter] = *input2;
counter++;  }
while((*input2 ^ check_for_line_feed) != 0);

frame_array[counter-1] = 0x0a;
counter = 0;
char * check_me = &frame_array[0];
float frame_size = atoi(check_me);
GOP_Array[GOP_position] = frame_size;
GOP_position++;       }

        float B_Sum = GOP_Array[1] + GOP_Array[2] + GOP_Array [4] + GOP_Array[5]+ GOP_Array [7] + GOP_Array[8] +
        GOP_Array [10] + GOP_Array[11];
float P_Sum = + GOP_Array [3] + GOP_Array[6] + GOP_Array [9];
float B_PR = B_Sum / P_Sum * 0.375;

if(GOP_position = 0) BP_average = B_PR;
else BP_average = (BP_average + B_PR) / 2;

BP_SetData(B_PR);                                   // Log the B_PR
GOP_position = 0;
big_count++;  }

AfxMessageBox("Writing");                           // Write the B_PR to a file called rosefile

BP_OutputFile = CreateFile( "d://MPEG Files//rosefile.txt", GENERIC_WRITE,
FILE_SHARE_READ, NULL,
CREATE_ALWAYS, 0, NULL );
if (BP_OutputFile == INVALID_HANDLE_VALUE)
{AfxMessageBox("Could not open Output File"); exit(0);}
BP_SaveData();
CloseHandle( BP_OutputFile );
BP_OutputFile = INVALID_HANDLE_VALUE;
CloseHandle( input_file );                          // Exit neatly
input_file = INVALID_HANDLE_VALUE; }
```

4.2 FrameReader.h

```
#include "afxmt.h"
class DiskReader;

class FrameReader : public CFrameWnd
{public:
int I_Frame_Total;
int B_Frame_Total;
int P_Frame_Total;
int current_type;
float GOP_I_Frame_Total;
float GOP_B_Frame_Total;
float GOP_P_Frame_Total;
float bp_ratio;
int bp_calc_first;
FrameReader();
virtual ~FrameReader();
BYTE * MPEGFileBuffer;
void OpenFile();
HANDLE m_hFile;
BYTE LookAheadBuffer[6];                            // Frame header = 4 bytes then 1 byte then frame type byte
UINT MPEGFileBufferEnd;
UINT MPEGFileBufferPosition;
BOOL First;
```

```
void CountFrames();
UINT GetMacroFrame(BYTE*pMacroFrame,BOOL*LastBuffer); // Gets a macroframe from the file/buffer
int counter;
UINT Frames;
UINT GetFrame( BYTE * pFrame );                    // Gets a frame from the file/buffer
BOOL FileFinished;
HANDLE m_hOutputFile;
void SetData( DWORD data );
DWORD DataArray[ 200000 ];
DWORD position;
void SaveData();
void WriteData( DWORD size );
HANDLE BP_OutputFile;
void BP_SetData( float data );
float BP_DataArray[ 5000 ];
DWORD BP_position;
void BP_SaveData();
void BP_WriteData( float size );
void Calculate_BP_Ratio();
void ResetMPEGFile();
BYTE * FindNextStartCode( BYTE * FramePTR, UINT * FrameSize, UINT MaxFetch );
BYTE * MoveOffCurrentStartCode( BYTE * FramePTR, UINT * FrameSize );
int FindElement(int * element);
CCriticalSection ConflictSection;
BOOL Locked;
BOOL FillNextBuffer;                               // Used for Macroframing - Defines to return 0 and show that
                                                   // file has been finished, and then to fill the next buffer available
                                                   // to be ready next time there is a request for a play by the client

DiskReader * disk;
static UINT StartDisk( LPVOID pParam );
void SetEndOfMPEGBuffer( DWORD Position );
void SetFileFinished();
};
```