MEDIA STREAMING PROTOCOL: AN ADAPTIVE PROTOCOL FOR THE DELIVERY OF AUDIO AND VIDEO OVER THE INTERNET

BY

CHRISTOPHER KARL HESS

B.S., Tufts University, 1991
M.S., Tufts University, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

Abstract

The introduction of audio and video content on the World Wide Web has greatly increased traffic on the Internet. The large size of these files results in long download times and may result in excessive start latencies. Video streaming has been used to reduce this latency. Frames must be delivered in a timely fashion due to the real-time nature of audio and video. Congestion within the network may delay the reception of frames and cause them to miss presentation deadlines. Several techniques have been developed previously to deal with congestion as the media is being transmitted. A new protocol, the Media Streaming Protocol (MSP), has been developed to stream audio and video over the Internet, which is fair in its consumption of network bandwidth. The protocol incorporates a novel technique to identify congestion within the system. The algorithm monitors the state of the system and is able to quickly adjust to the changing capacity of the network. A feedback mechanism is used to notify the server to take corrective action in the presence of congestion. The protocol reduces the amount of data being transmitted and minimizes the aural and visual defects during playback. Such a protocol is critical as network bandwidth is increasingly being consumed by audio and video applications.

Acknowledgments

There were several people that helped me during the development of the protocol. I would like to thank Roy Campbell for giving me the opportunity to freely develop and experiment with different designs. The discussions I had with Dave Raila provided me with ideas and directions to pursue. I would like to also thank See-Mong Tan and Zhigang Chen, the creators of the original adaptive feedback protocol, VDP, for their help and design inspiration. And last but not least, I would like to thank Anda Harney for all-around administrative help.

Table of Contents

1 Introduction

With the explosion in popularity of the World Wide Web (WWW), network traffic has been steadily increasing and overloading the capacity of many network connections. The large size of images and video sequences are particularly to blame for much of this traffic. Another problem with the size of these media forms is the latency experienced by the user in downloading these files. Video sequences can be very large (on the order of megabytes) and may take unacceptably long periods of time to transmit. One technique to reduce this latency is to allow the viewing of frames in the sequence before the entire sequence is transmitted. This technique of "video streaming" has become a popular alternative to waiting for the complete sequence to be downloaded before the first frame may be viewed.

In this thesis, a new protocol called the Media Streaming Protocol (MSP), has been developed to stream video and audio data over the Internet. The protocol is designed to adapt to congestion when it occurs within the network. A feedback loop is utilized to relay information from the client displaying the media to the server sending the media to signal when congestion is being experienced. When congestion has been noticed, the server will regulate the flow of data in such away to attempt to relieve the congestion and allow the client to display the data with a minimum of visual and aural defects. The flow control mechanism may result in temporary degradation in playback quality when connections do not have enough bandwidth to accommodate all the data traffic. This temporary condition is an attempt to be fair in its consumption of network bandwidth. Client side buffering is employed to smooth the jitter of arriving data packets and to place packets that arrive out-of-order in the correct location in the sequence. Lost packets may be retransmitted if it is determined that sufficient time remains to retrieve the frame before it must be presented.

The thesis is organized as follows: section 2 describes some of the previous work that has been done in the field of video streaming. Section 3 discusses some of the issues that arise in the streaming of media data. Section 4 describes the design of the protocol and section 5 explains the implementation of the protocol and players in more detail. Section 6 explains the simulation of the network used for the experiments and section 7 presents the results of these experiments. Section 8 discusses some possible directions for further research and section 9 presents some concluding remarks. The appendix presents class diagrams of the design used in the implementation.

2 Previous Work

Much work has been devoted to the field of video streaming. Many different approaches have been taken in the delivery of video data, depending on the working environment. Most of the effort has focused on avoiding or reacting to congestion within the network. Both lossless and lossy techniques have been developed. In addition, some techniques attempt to provide guaranteed service, while others rely on best-effort service networks. Schemes that guarantee Quality of Service (QoS) often try to prevent congestion from occurring by allocating the resources necessary for the transmission of the video data. In contrast, many best-effort methods react to congestion after it occurs within the network and must take appropriate action.

Lossless techniques require that no information be lost during the transfer of the video sequence and that the quality of the video sequence not degrades. These methods ordinarily use reliable transport protocols or operate on highly reliable networks, such as asynchronous transfer mode (ATM) networks. Many video compression techniques create variable bit rate (VBR) video sequences. Such encoding creates inherently bursty traffic when transmitted over a network and complicates resource reservation. Schemes have been developed that attempt to smooth the transmission of video sequences by producing segments of constant bit rate (CBR) traffic [1, 2]. Instead of sending frames at regular intervals, groups of frames are sent at a constant rate. This allows large frames to be sent in more time than the period of the sequence and smaller frames to be sent in less time. The effect is to reduce the level of burstiness in the network, which has been shown to produce less congestion. Video smoothing has also been applied to the transmission of real-time video sequences, such as video teleconferencing and live broadcast [3, 4].

Other lossless techniques have been developed to guarantee end-to-end delays in local area networks (LAN). Such methods involve incorporating admission control when accepting new connections [5, 6, 7]. Other protocols have been designed to increase the number of acceptable connections by periodically renegotiating for the currently needed resource requirements [8]. Yet others have relied on the law of large numbers to reap the benefits of statistical multiplexing for a relatively large number of uncorrolated streams [9]. Further, it has been demonstrated that increasing the delay (through buffering of data) can increase the statistical multiplexing gain in the system [10].

Streaming over the Internet is characterized by best-effort service, since the Internet protocols (TCP and UDP) provide no hard-time guarantees for timely delivery of data. This makes lossless schemes more difficult. Some lossless techniques deliver frames using a reliable protocol and buffer a number of frames at the client. Increases in congestion may deplete the buffer and the playback of the video sequence must be stopped for some time to allow more frames to be buffered [11, 12, 13]. In order to reduce the risk of buffer underflow, an adaptive scheme it necessary. Such lossy techniques often utilize a feedback mechanism to notify the sending machine to reduce the amount of data being sent. The sending machine may drop frames in the sequence or send frames with degraded quality (i.e., changing the quantization of encoded frames). Real-time capture applications may require that the encoder vary the output based on such feedback [14, 15, 16], while pre-recorded video sequences may require storage of multiple video sequences of varying quality. Such lossy schemes may not send all the data in the sequence and may provide temporary degraded playback for periods of time. A TCP-like mechanism called the Streaming Control Protocol (SCP) has been developed where each packet sent is explicitly acknowledged and expired timers trigger packet retransmission [17]. Several schemes have been developed in which the client monitors the state of the system and notifies the server via a control channel to thin the stream of video data transmitted. A distributed real-time MPEG video and audio player was developed which incorporated an explicit feedback loop to control the sending of frames [18]. The user may specify an initial frame rate. The algorithm compares the user-specified frame rate, the target frame rate, and the display frame rate, and notifies the server when necessary to drop frames from the stream. The Video Datagram Protocol (VDP) was developed for use in a modified version the Mosaic web browser, called Vosaic (Video Mosaic) [18, 20]. Vosaic extends standard HTML syntax by including tags for audio and video streaming. The VDP protocol uses the arrival time of packets to determine the current state of the system. VDP supports corrections for congestion in both the network and CPU processing power, as well as retransmissions of lost frames. The protocol was later incorporated into a plug-in for use in commercial browsers.

The protocol presented in this thesis is an adaptive lossy protocol, which attempts to deliver the media data in real-time. The protocol was designed to operate over the Internet, which is highly dynamic and only provides best-effort service with no guarantees in QoS. A novel technique has been developed to identify congestion within the network. The protocol reacts to congestion after it occurs and makes effort to alleviate the congestion by judiciously dropping packets. The proto-

col is designed to be fair in its use of bandwidth and to present the audio and video to the user with a minimum of degradation in quality.

3 Issues in Streaming Audio and Video

The real-time nature of audio and video data introduces complications that must be considered when streaming these forms of data. These complications relate to the method of transmitting the data, the characteristics of particular media types, and the handling of multiple streams.

3.1 Transport Protocols

Data that is not restricted by the time at which it must be delivered is defined to have soft deadlines and allows more variability in the acceptable delivery schedule. For example, in transmitting files (i.e., using the FTP protocol), the most important criterion is that the file be received without error. In such a case, reliability is more important than timely delivery (although it should not be delayed for too much time). The underlying protocol must make effort to deliver all data correctly and in order. A reliable protocol, such as the Transmission Control Protocol (TCP), is required as the underlying transport mechanism. The guaranteed delivery of data, however, comes at the price of possible delays in the reception of data; multiple retransmissions of data packets create unbounded delays in delivery. Subsequent packets may not be delivered to the user application until previously missing packets arrive. This delay may be unacceptable for real-time data delivery.

Real-time data such as audio and video, on the other hand, require the delivery of data in a timely fashion and must meet hard deadlines. In this case, receiving data frames before the deadline is more important than reliable delivery. That is, a late frame is equivalent to a frame that never arrives, since it can no longer be displayed and is worthless. An unreliable protocol is acceptable and desired in this situation, which makes no guarantees for the delivery of data. Data may be delayed, out of order, or even missing. An example of such a protocol is the User Datagram Protocol (UDP). UDP makes no guarantees regarding the order and completeness of the data being transmitted. Transport schemes built upon unreliable protocols may take some action to provide some degree of recovery of lost data, but may not require the complete recovery that a reliable protocol guarantees. A reliable protocol in such situations would waste bandwidth because late frames would be retransmitted even though they cannot be used.

3.2 Media Forms

Digital audio and video sequences are often quite large and must be compressed to reduce both transmission times and disk storage space requirements. Depending on the compression technique, differing ratios of compression may be achieved. Further, the technique defines if the produced frames will have differing or similar sizes. Many video compression techniques produce VBR frame sequences, in contrast to many audio compression techniques, which produce CBR frame sequences.

3.2.1 Video

Several video compression techniques (MPEG, H.263) recognize that there is redundant information between video frames (statistical redundancy), as well as information that may not be visible to the human eye (perceptual redundancy) [23, 24]. Statistical redundancy may be exploited by encoding frames as differences from the next frame, previous frame or both. Perceptual redundancy can be used to further reduce frame size by eliminating information that cannot be perceived by the eye.

MPEG (Motion Pictures Expert Group) encodes three types of frames to exploit statistical redundancy; I (intra), P (predictive), and B (bi-directional predictive). I frames have no dependency on other frames and may be decoded without reference to any other frame. In contrast, P frames are encoded as differences from the previous frame, and B frames are encoded as differences from the previous and next frames. I frames, therefore, are the most important type of frame, because all subsequent frames, until the next I frame in the video sequence, rely on it for proper decoding.

Because P and B frames use information from neighboring frames, they are usually significantly smaller than I frames. B frames are usually the smallest, using information from two surrounding frames. Typical B frames are 10x smaller than I frames. This variability of frame sizes has implications when they are being streamed across a network, which will be discussed later.

Sets of I, P and B frames are grouped together in repeated sequences known as a Group of Pictures (GOP). Therefore, the video sequence contains a fixed ordering of frame types. As will be discussed later, the knowledge of this predictable sequence of frame types is important for the protocol when it must adjust to network congestion.

3.2.2 Audio

Many audio techniques create frames that are identical in size (μ-law, GSM). This is due to the fact that there is less information in surrounding frames that can be used to reduce the size of a frame.[1] GSM is a low bit rate audio compression technique that encodes digital audio at 1650 Bps. The μ-law compression format was developed for voice quality audio. Compression ratios of 8000 Bps are achieved for use in digital telephone systems. The CBR nature of many audio encoding techniques makes it easily adaptable to transmission over networks.

3.3 Issues in Streamed Media

The variability in frame size of MPEG encoded video sequences produce VBR traffic when the frames are sent at regular intervals over a network. This makes the transmission of MPEG videos inherently bursty. For example, suppose a video sequence is being transmitted at 10 frames per second (one frame every 0.1s) and the size of an I frame is 100,000 bits and a B frames is 10,000 bits. The network capacity needed to send an I frame in 0.1s is 1.0Mbps, while a B frame requires only 0.1Mbps.

A router is in charge of accepting packets, determining where the packet should be routed during the next hop, and sending the packet. The UDP protocol maintains a receive buffer in the operating system kernel for each connection (port) that is used to hold queued packets. If the output line has much lower bandwidth than the input line, there is a possibility that the receive buffer may overflow. If a packet arrives and there is no room in the receive buffers, UDP will silently drop the packet. When packets are dropped, the network is said to be congested, because the router cannot service the packets at the speed at which they are arriving.

[1] This is not true of differential audio encoding techniques.

Congestion occurs when more data is being sent than the network can handle. If the network is near its capacity and a new connection is established, the sum of the connections may exceed the network's capacity. With VBR traffic, it is possible that a connection may cause congestion for itself, regardless of the other connections. For example, say each MPEG frame is being sent in one UDP packet and they are being sent at a specified rate, one frame every *x* seconds. If a large packet arrives at a router and the output bandwidth is small, it may take much longer than *x* sec-

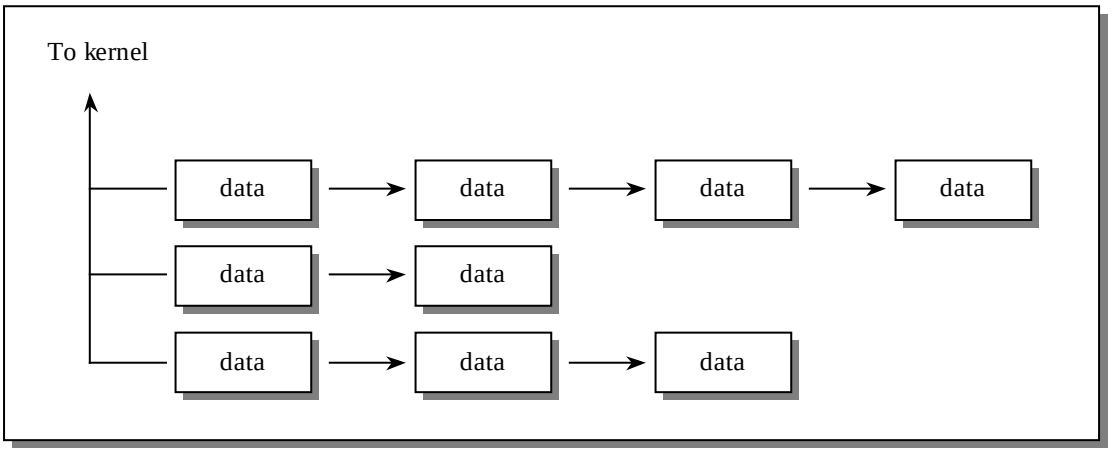

Figure 1 Diagram of the UDP receive buffers for different ports. If too many packets arrive at a single port, packets will be silently discarded.

onds to send the packet. During the time that the packet is being sent, packets are arriving and being stored in the receive buffers [21, 22]. It is possible that the large packet may take so long to transmit that the receive buffer becomes full and arriving packets get dropped. In addition to dropping packets, UDP may delay or reorder packets.

It is also possible that the server sending the packets may not be able to send the packets at the prescribed rate, because too many connections have been established. The problem of processor speed can also arise on the receiving side of the protocol. If the processor cannot perform fast enough for the protocol to work correctly, performance of the protocol may degrade. The protocol must decide how to handle each of these situations.

3.4 Multiple Streams

The differing sensitivity of human hearing and sight indicates that audio and video should be handled differently when adverse conditions arise that affect the playback of media streams [25]. It is well known that the aural senses are much more sensitive to disturbances than the visual

senses. Therefore, it is appropriate to give priority to audio data over video data. If there is congestion in the network and any data is discarded, it is preferable to discard video data first.

There are several methods when transmitting multiple related media streams. One method is to send the audio and video components in one stream, interleaving the two components. This has the advantage that the synchronization of the streams for playback is not needed, since, for example, one second of video can be interleaved with one second of audio. Another method is to send the audio and video components in separate independent streams. For a protocol that wishes to adapt to network congestion, this option is more appropriate. As described above, it is desirable to give priority to audio, which would be impossible if the streams are integrated into a single stream. When a period of congestion is experienced, it is possible to drop some video frames to relieve the bottleneck, while allowing the audio to flow unaffected. There may become a point where congestion is too great and some audio frames may need to be dropped, but this should be only considered after a greater number of video frames have been dropped. When using this approach, however, care must be taken to properly synchronize the audio and video streams during playback. Excessive skew (greater than 80 ms) between audio and video streams can cause noticeable synchronization errors [25].

4 System Design

The following sections give a brief overview of the design of the protocol. Later sections will discuss the system in more detail and provide a description of the implementation.

4.1 Control and Data Channels

The MSP protocol uses a point-to-point client-server architecture, as shown in Figure 2. The protocol consists of two socket connections; a data channel to send the data frames and a control channel that allows the client to communicate with the server to relay messages regarding the current status of the network. This feedback loop (control channel) lets the server take appropriate

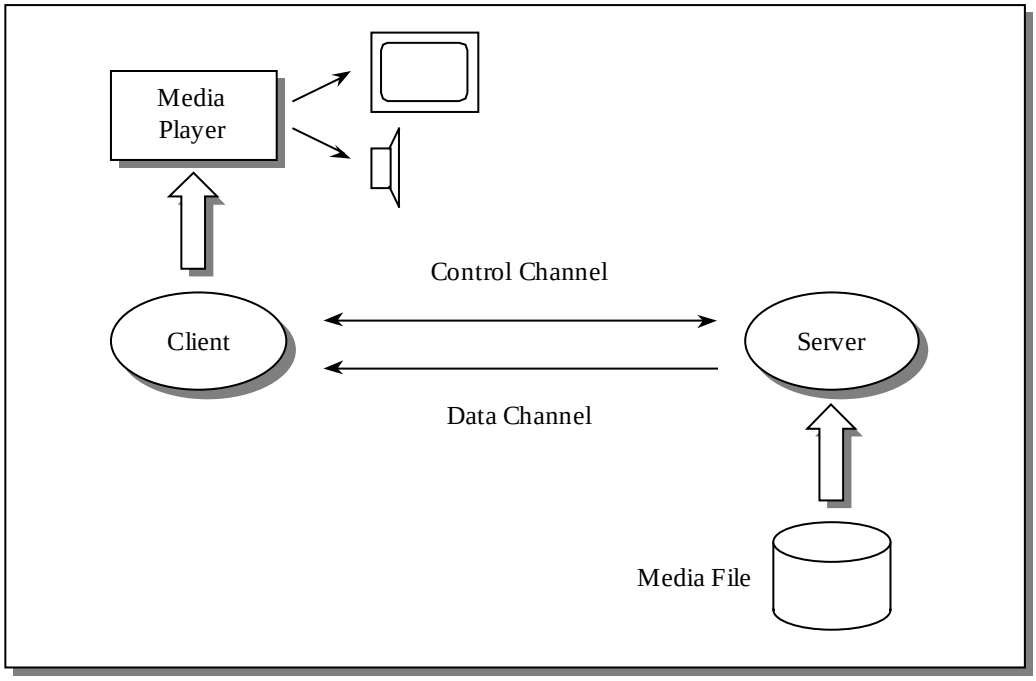

Figure 2 Diagram of client-server architecture for MSP protocol. The server reads the media data from the file and transmits data packets on the data channel. The client places the data into the buffers until it is read by the media player. The control channel is used to relay information from the client to the server to notify it of problematic conditions within the system.

action in response to requests from the client. The control channel is also used during the handshaking phase when the protocol is being initialized, and for sending stream header information, e.g., the MPEG header. The control channel is a reliable two-way connection using the TCP protocol. The data connection uses UDP datagrams to send packets to the client. This configura-

tion reduces the delay experienced by data packets and allows the reliable exchange of control and handshaking information.

4.2 Data Flow

When the client receives packets of data, they are placed in buffers maintained by the protocol. The buffers allow for random placement, so out-of-order packets may be placed in the correct location in the buffer. Every time some data is read by an application, the protocol checks the status of the buffers to see if the number of frames in the buffer is increasing, decreasing, or stable. When ideal conditions are present (there is no congestion and the server is sending out packets at the correct rate), the number of frames in the buffer should remain constant. If it is determined that there are abnormalities in the network or server, the client sends messages via the control channel to the server to adjust the flow of data to rectify the condition. The protocol does not use acknowledgments; it uses feedback from the client (essentially negative acknowledgments) to request reductions in data and retransmission of lost frames. This has two advantages in a time critical environment. First, the amount of traffic flowing on the network is limited and second, it allows for greater scalability by relieving the server from the burden of the more complicated management of acknowledgments.

4.3 Adaptive Algorithm

Since the data must be presented in a timely fashion, it cannot be delayed for any significant amount of time. Measures must be taken when the delay of data becomes too great. If the network becomes congested, an attempt is made to relieve traffic congestion by decreasing the amount of data that is being transmitted. This is achieved by dropping audio and video frames. As discussed in section 3.3, video frames are dropped before audio frames, due to the difference in sensitivity of human vision and hearing.

A determination must be made as to when problematic conditions have occurred. The protocol monitors the buffer of frames that it has received, but that have not been read by an application for playback. A moving set of high and low watermarks is used in this buffer to determine current conditions. More specifically, a problem is indicated when the number of buffered frames falls below the current low water mark. When this occurs, a message is sent to the server via the control channel to take corrective action. The low and high watermarks are then adjusted downward. If the number of frames falls below the new low watermark, another message is sent. Similarly, a message is sent to the server each time a high watermark is passed, indicating that the

level of severity of current conditions has been reduced and the server can increase the amount of data it is sending. The messages that are sent to the server vary depending on the particular problem in the system.

5 Implementation

The purpose of developing the MSP protocol was to enable the adaptive transmission of video and audio data over the Internet. Much of the traffic over the Internet, especially audio and video traffic, is initiated through web browsers. Therefore, it would be appropriate for the protocol to operate in current web browsers. In order for executable code to be initiated from a web browser, it must be either incorporated as a "plug-in", which must be downloaded by the user, or be written in the Java programming language [26].

It was decided that, with the introduction of the Java Media Framework (JMF) in future versions of web browsers and the increasing performance of the Java virtual machine, implementing the protocol in Java would be beneficial. Currently, JMF is not fully mature and was not used. However, the protocol may be used in the JMF framework when all the necessary functionality is supported. The MPEG software decoding engine used is a modification of Berkeley's mpeg2play software written in C and has been ported to several different platforms (UNIX and Windows NT). A Java interface to the native MPEG decoder methods using the Java Native Interface (JNI) was developed. Future work will incorporate JMF and will thus eliminate the need for any native code. The Java Audio API was used for audio playback.

Since the protocol operates over a network connection, a client-server architecture was adopted. The server is responsible for accepting new connections and sending the video frames at regular intervals. The client is responsible for monitoring the state of the connection and notifying the server to adjust transmission if necessary. Video and audio players may then read the data from the client for playback of the media data.

Each section below describes different aspects of the implementation. The implementation is described by tracing the flow of data through the system. The protocol is explained first, including the client and server sides, followed by a description of the media players used. The class name used in each part of the implementation is specified in the parentheses in the heading of each section; these can be used to match functionality of the components to the UML diagrams [30] presented in the appendix. Several design patterns were used in the implementation to provide a clean and structured design [27, 28, 29].

5.1 Client Side

As described above, the client side of the protocol is responsible to monitoring the state of the network, client and server. If problems in any of these components are realized, the server is notified to take corrective action to relieve the problem. The client side performs all of the logic needed to determine if something has gone wrong. The server merely responds to the information provided by the client. This allows the server architecture to be relatively simple and provides greater scalability. Therefore, the server can handle more connections, because it will not be burdened by complex algorithms.

The flow of data packets will be used to describe each of the functional units needed and their responsibilities. The system is heavily threaded. However, there are two main threads that do most of the work on the client side. The first thread waits for packets from the network and stores them in its internal buffers. The second thread is that of the player, which reads the data from the buffers and presents it to the user. The flow of data entering from the network will be described first followed by a description of what happens when data is read by an application.

5.1.1 Packet Structure (`MSPDatagram`)

Data is transmitted in packets, which contains a header and payload portion. The header contains information needed by the client to correctly identify the data contained in the payload and is needed by the protocol for correct operation. The header, shown in Figure 3, is 10 bytes long.

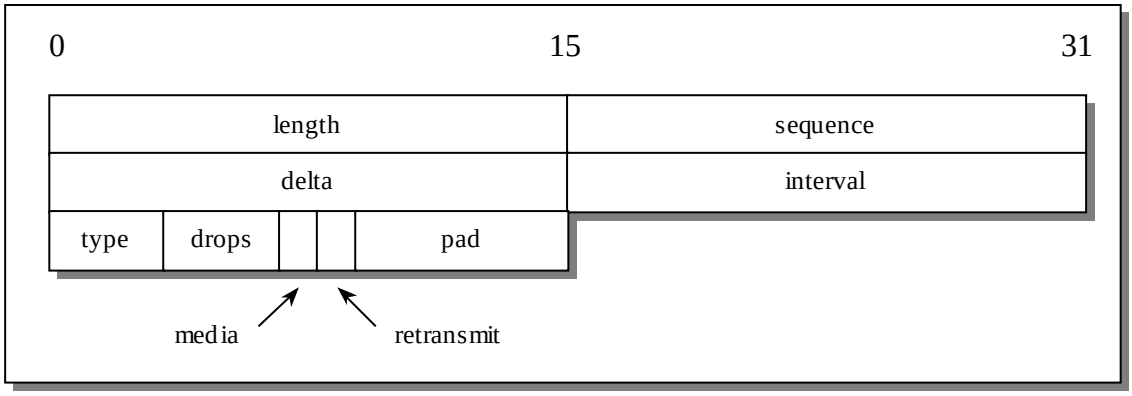

Figure 3 Description of packet header.

The *length* field (16 bits) specifies the length of the payload in bytes. The *sequence* field (16 bits) identifies the location of the packet in the data stream. The *delta* field (16 bits) describes the difference between the original time interval between packets that the server was sending and the

current interval at which the server should be sending packets. The *interval* field (16 bits) is the actual time between data packets being transmitted by the server. The *type* field (4 bits) describes the type of data specific to the particular media type (i.e., I, P, or B frames). The *drops* field (4 bits) describes the number of drop requests that have been received by the server. The *media* field (1 bit) identifies the type of media data in the payload (audio or video). The *retransmit* field (1 bit) tells whether the packet is a retransmitted packet. The *pad* field (6 bits) aligns the header to the byte boundary.

5.1.2 Clocks (`Delay`)

Applications need to display video frames at regular intervals and frames must be sent over the network at a regular interval. It is necessary to have an accurate method of specifying an amount of time to delay execution while waiting for the interval to expire. Successive calls to

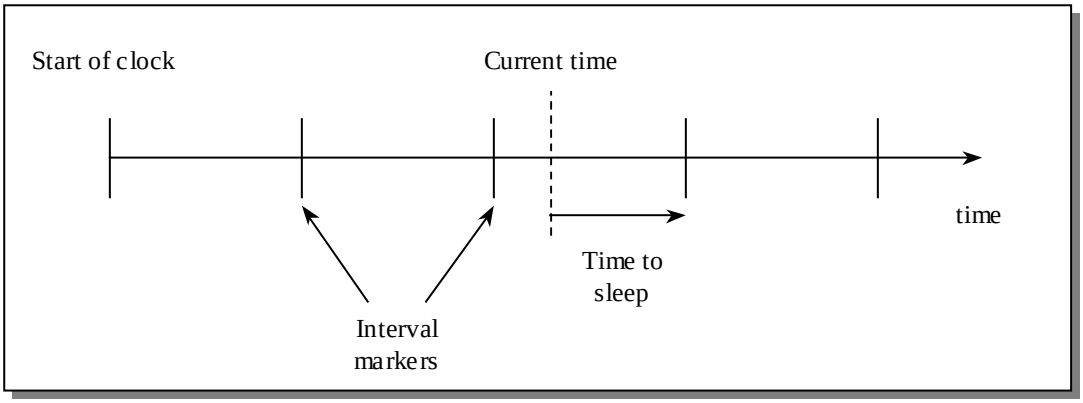

Figure 4 Description of clock that measures intervals more accurately. The next target interval is subtracted from the current time. The thread then sleeps for that period of time. The target interval marks are measured from the start of the clock.

`Thread.sleep()` may result in accumulated errors which can give inaccurate delay values over time, since the method is only accurate on the order of milliseconds (~10 ms) on most operating systems. A generic delay object was created to give better results that will stay accurate over time. Delay values are always calculated on an absolute time scale. The end of the current sleep period is measured from the start of the clock instead of from the last time the call to sleep was made. Although delay values are only as accurate as the underlying operating system allows, this error will remain fixed for successive calls. This is especially important for matching the rate at which the data is received off the network, the rate at which frames are read from the displaying application, and the rate at which the server is sending packets. If these rates do not match, the size of the buffer will fluctuate and the protocol will not operate correctly. An alternate option

would be for the delay function to loop continually until the specified delay was reached. This busy approach gives more accurate results (~ 1 ms), but requires excessive CPU cycles.

5.1.3 Transmission of Media Data (`Transport`)

Packets are transported over the network via a transport object. A simple object encapsulates the methods needed to create and destroy a UDP connection, as well as to send and receive packets over the connection. The underlying transport protocol could be any protocol that is deemed acceptable. UDP fills the needs of real-time transport, but may be replaced if desired.

5.1.4 Stream Creation (`DataSource`)

Once a media stream is requested, a connection to the server must be established. When the server has been contacted, the client's host and port are transmitted to the server so that UDP packets may be sent to the correct destination. The server next sends some initialization information over the control channel needed to create the correct client side engine (media type and format), described below. Once the stream has been established, the client can begin to accept data packets from the network.[2]

5.1.5 Client Side Engine (`MediaManager`)

The client side engine is the component that manages the data buffers, network monitor and access to the control channel. The engine accepts packets from the network and stores them into the buffers. It also allows higher level applications to read data from the buffers. It is responsible for determining when problems have occurred in some component of the system, be it the network, client or server, and how to correct the situation. Most of the tasks it must perform are delegated to other objects. These objects are discussed in the following sections.

5.1.6 Packet Buffering (`MediaBuffer`)

Once a data packet has arrived, it must be stored until an application reads the data. Buffering the data allows the protocol to smooth the jitter in the arrival of packets and allows packets to be given to an application in the correct order if they arrived in incorrect order. Since it may be necessary to place the packets in a different order from how they arrive, the sequence number of the packets must be preserved. However, it is important that data be read quickly once it has been

---

[2] The `DataSource` class extends `javax.media.protocol.PullDataSouce` as required by JMF. Although JMF is not currently being used in the implementation, the class hierarchy follows the JMF specification so that the protocol can be easily incorporated into the JMF framework.

requested. The buffer has been implemented as two buffers, one for data and one for sequence numbers. Once a packet arrives, the header is stripped off and the necessary components are stored in the appropriate buffer. This allows for quick retrieval of data, while retaining the correct ordering of packets within the buffers.

The object encapsulating the buffers provides methods for writing and reading to and from the buffers. When frames are read from the buffers, it must be decided whether the desired frame is present, has been dropped by the protocol, or never arrived. When frames have been dropped by the protocol, the packets contain only header information and no payload data. As will be described later, this is important for the correct retransmission of missing packets. Correct frames are simply returned to the requester. A reference to the last correct packet passed to the requester is maintained to deal with dropped and missing frames (an empty packet is maintained in the case of audio).

Since an application requesting data is expecting to get frames at a controlled rate, some data must be passed for each request. (The design does not allow the player to pause until the next correct frame is available.) In the case of missing and dropped frames, the data of the last correct frame is used as a placeholder. For example, if frames 100 – 105 do not arrive, frame 99 will be supplied for frames 99 – 105. While frames 100 – 105 are being read, the low sequence number in the buffer is adjusted accordingly, but frame 106 would be the first frame in the buffer. Although this method of returning the same frame multiple times may give some visual or aural defects in playback, it is necessary to keep the real-time nature of the streams in tact, as well as synchronization between the streams. However, the protocol attempts to minimize these defects by regulating the amount of data being transmitted and therefore reducing the number of frames dropped by the network. Section 5.2.5 will explain how the protocol is able to reduce the visual degradation.

Another alternative would be to stall the player if a frame was found to be missing or dropped. This would create a tight coupling between the protocol and the player, which would hamper the ease of using different playback software with the protocol.

Before a packet is placed into the buffers, the client side engine checks to see whether it is the expected packet. If it is not, it signals that the network has dropped a packet and a request is made to retransmit the frame. This request is only issued if it is estimated that the retransmitted

frame would arrive soon enough to be displayed. The implementation allows frames to be retransmitted only once; multiple requests to retransmit packets would increase the amount of congestion in the network.

5.1.7 Round Trip Time Calculation (`RTT`)

In order to determine whether a retransmitted packet could arrive before the deadline for displaying the frame, an estimate of the round trip time is needed. This time is the measure of how long it would take to send a request and receive the frame. At regular intervals (every 3 seconds), a message is sent via the control channel. The current time is attached to the request. Once the server receives such a message, it is immediately returned to the client. The client can then subtract the current time from the timestamp in the message. A low-pass filtering function is used to slightly smooth the value using past values:

$$RTT = \alpha\, RTT + (1 - \alpha)\, RTT_{old} \qquad (1)$$

where $0 \leq \alpha \leq 1$. The variable α is the degree of weight given to old and new values. In the implementation, α was set to 0.8.

The round trip times are a bit optimistic, since the packets being sent for the calculation are much smaller than actual data packets, which may take more time to transmit. However, the round trip time calculation is only an estimate, since the current network bandwidth can be highly dynamic and therefore is an acceptable estimate for the current state of the network.

5.1.8 Sending Control Messages (`Controller`)

Any messages that must be guaranteed to reach the server from the client are sent on the control channel. Handshaking messages sent during protocol initiation are also sent in this channel. The control channel uses TCP as the transmission protocol to guarantee delivery. All command messages are sent as text strings (Java UTF), similar to many other existing protocols, such as FTP and SMTP. An object was created to encapsulate the sending of all possible control messages. The possible messages include:

1. MSP_FRAME retransmit a particular frame
2. MSP_RECV initiate connection handshake
3. MSP_DROP drop some number of media frames to the stream

4. MSP_ADD — add some number of media frames to the stream
5. MSP_PERIOD — change the sending period between frames
6. MSP_RTT — round trip time timestamp
7. MSP_TRANSFER — start transmitting data
8. MSP_HOST — host to send data to
9. MSP_PORT — port to send data to
10. MSP_STREAMS — number of streams to send (1 or 2 for audio and/or video)
11. MSP_ID — unique identifier used to synchronize start of streams
12. MSP_STOP — stop transmission

This object can both send and receive messages.

5.1.9 Gathering Statistics (`Statistics`)

Once a packet has been received and placed into the buffers, some statistics must be gathered that are needed for operation of the protocol. The protocol uses relative timestamps of certain measured intervals to monitor how the system is operating. The three intervals are the packet arrival interval, sending interval, and reading interval. These are the time interval between successive packets arriving at the client, sent by the server, and read by an application, respectively. Each of the interval values is measured as an average of the last 5 values collected (this gives more stable values when conditions are in flux). The use of these values will be described below in section 5.1.11. Information from retransmitted frames is not used, since they are not sent at any regular interval.

The previous sections described the flow of data as it enters the client. Once the data is in the protocol buffers, it may be read from an application for display to the user.

5.1.10 Buffer Access Regulator (`Regulator`)

When an application wishes to display frames, it must read the encoded data, decode it into raw frames and present them to the user at regular intervals. Many video decoders don't read one frame at a time, but rather read fixed amounts of data. Allowing the decoder to read the data in this fashion would cause the size of the buffer holding data within the protocol to decrease rapidly. As the decoder is decoding frames and the application is displaying the frames, the buffer

size increases as data packets are arriving from the network. This would continue during the playback of the video sequence, creating an oscillation in the size of the buffers.

As will be described in the next section, the proper operation of the protocol depends on the system being in steady state, i.e., frames should be read from the buffers at the same rate as they arrive from the network and are written to the buffers. If any decrease in the size of the buffer is noticed, it must signify some problematic condition in the system, not a request for data from the decoder. Therefore, it is necessary to regulate the speed at which the decoder is allowed to extract frames from the buffer. A mechanism is installed within the `read()` method, which regulates the rate of read method calls. Every time data is read, the protocol suspends the calling process for an amount of time, so data is accessed at intervals equal to the period of the video sequence display. In addition, only one frame of data may be read at a time. For example, if the video sequence frame rate is 10 frames per second, each read call would be suspended for 100 ms from the previous call. Each read call would only be allowed to copy one frame of data to the decoder, not the complete buffer that the decoder requested.

The regulating mechanism can be thought of as a token bucket, where $\rho = 1/\text{fps}$, $\beta = \infty$ (no upper bound on the bucket size). Therefore, if the decoder is stalled for some reason, the regulator can accumulate credits at a rate of one credit for each period of time elapsed. When the decoder does finally issue the read, it may read as many credits worth of frames as have accumulated. In this way, the decoder will not deviate from the given frame rate of the video sequence. The regulating mechanism uses the clocking mechanism described in 5.1.2 for accurate time measurements. The end result is that during correct operation of the protocol the number of frames in the buffer should remain stable.

5.1.11 Protocol Monitor (`BufferTracker`)

The protocol must determine when problems in the system have occurred. This can be due to network congestion, a slow server, or a slow client. The protocol uses the changing size of the buffer over time to determine the current state of the system. This offers a more stable reference than simply measuring the inter-arrival time of packets. If the buffer size starts to decrease, some problematic condition has occurred. Similarly, an increase in the size of the buffers signals relief of the condition.

One method to accomplish this task would be to set a high and low watermark in the buffer. When the buffer size decreases below the low watermark, action would be taken to rectify the problem. The problem with this approach is that it can take action only one time. If the system problem persists and the buffer size continues to decrease, no further action can be taken, since

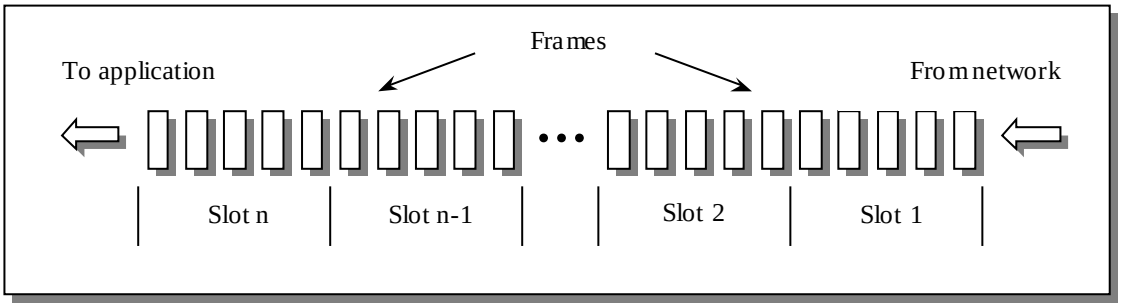

Figure 5 Diagram of logical slots in buffer. The buffers are divided into slots, which the protocol uses to determine the state of the system. High and low watermarks are used to determine when problems have occurred in the system.

the low watermark has already been passed. Therefore, the protocol uses a moving set of high and low watermarks. The buffer is logically broken into *slots*, which represent a fixed number of frames (See Figure 5). The boundaries of the current slot mark the current high and low watermarks.

Three values may be specified to the protocol to tune the operation; the *buffer time*, *slot time*, and *check time*.[3] When the protocol is first started, a number of frames are prefetched and buffered before an application is allowed to read them. The amount of time that frames should be prefetch may be specified to the protocol and is called the *buffer time*. For example, if the frame rate is 10 frames per second and the amount of time to prefetch is 4 seconds, 40 frames will be prefetched. The high and low watermarks are initially set at the boundaries of Slot 1. If the number of frames in the buffer decreases so that no more frames are in Slot 1, the high and low watermarks are adjusted to the boundaries of Slot 2. This continues as long as the size of the buffer decreases. Similarly, if the size of the buffer increases above the high watermark, the high and low watermarks are adjusted up to the next slot. Once a new slot has been entered, it must be determined what caused the change in buffer size and the server is notified to take appropriate action. The size of the slots (and number of slots) may be specified to the protocol by specifying the *slot time*.

[3] Buffer time, slot time and check time are specified in milliseconds in the implementation.

The number of frames in a slot can be calculated by dividing the slot time by the period of the media sequence.

One additional parameter may be specified to the protocol: the *check time*. The check time tells the protocol at what point to start paying attention to decreases in buffer size (e.g., start checking decreases in buffer when the high and low watermarks are at Slot 3). As described in section 3.4, video frames should be dropped before audio frames since human hearing is more sensitive than vision. Therefore, the check time of audio stream should be lower than the corresponding video stream. For example, suppose the network becomes congested and both the audio and video buffers start to decrease. Once the video buffer decreases passed a low boundary and is lower than the check time, video frames will be dropped, but audio will not be touched, since the check time has not been reached yet. This will give the network time to adjust to the congestion and leave the audio unaffected. Using this mechanism, the same protocol can be used for both audio and video. The check time adds the ability to specify priorities for different streams.

In order to determine the cause of any problems in the system, the relative timestamps taken at several parts of the system are compared. The algorithm is based on the assumption that the relative rates of the clocks of different machines are fairly close, even though the clocks may not be synchronized. Timestamps are recorded each time a frame is sent from the server, a frame is received at the client, and a frame is read by an application. The timestamps are used to calculate the interval between successive sending, arrival and reading of packets. During proper operation of the protocol, each of the interval values should be identical. Each time a watermark is passed, the interval values are compared[4] to determine what caused the change in buffer size and what corrective action should be taken. When a low watermark has been passed (the buffer is decreasing), the following intervals are compared:

1. **Arrival interval > sending interval** tells if the network is congested. When the server is operating correctly, each packet should be sent at constant time intervals. If the arrival interval is less than the sending interval, congestion is signified, since the network is delaying packets. A message is sent to the server to drop frames from the stream.
2. **Sending interval > proper sending interval** tells whether the server is sending data too slowly. This may happen if the server CPU is overloaded. If the actual sending interval

[4] The relative difference of values must be greater than 20% in order for a difference to be recognized.

in greater than the interval at which the server should be sending packets, not enough CPU power is available. A message is sent to the server to drop some number of frames from the stream. This will reduce the time needed to send packets, since no data must be read from disk.

3. **Read interval > real frame rate of media** tells whether the client is slow. The frame rate at which the media should be delivered to the client is known. Each time an application reads a frame from the buffers, the interval is recorded. If this interval increases, the client does not have sufficient CPU power to keep up with the correct rate. Currently, no action is taken for this case in the implementation.

In the current implementation, an attempt is made to correct any problem by sending requests to the server to drop frames sent in the stream. Therefore, if the buffer is increasing in size and a high watermark is passed, a request to add frames to the stream is requested in all cases.

A question that arises is what action should be taken when it is found that the network has dropped a frame due to congestion.[5] Often congestion will result in several consecutive frames to be dropped by the network. As described in section 5.1.6, missing packets are retransmitted by the server. However, requests to retransmit packets will result in more data traffic and possibly more congestion. Should packets be retransmitted? The answer is "yes". It is better to let the protocol regulate what packets are to be dropped, not the network. In addition, any particular packet may only be retransmitted once. If it is dropped a second time by the network, the data in the packet is lost. As will be discussed in section 7, better results are obtained if the protocol decides what frames are to be dropped, since it has knowledge of what frames are more important for playback.

5.2 Server Side

The previous sections described the flow of data at the client. This section will focus on the server side of the protocol. The server is responsible for accepting new connections and responding to feedback from the client.

[5] Note that the network dropping frames is different from the protocol dropping frames. The network drops frames when there is congestion. The protocol drops frames in an attempt to relieve congestion.

5.2.1 Accepting Connections (`Server`)

The first task of the server is to accept new connections. Once a connection has been accepted, a new thread is spawned to handle the particular connection. A control channel is also established for reliable communication with the client. If more than one stream is required for the session (i.e., audio and video), it is necessary to synchronize the start of the two streams.

5.2.2 Synchronization of Streams (`StreamSynchronizer`)

All streams are sent via separate connections. It is necessary for the client to present the streams in a synchronized fashion to the user (described in section 5.3.4). The server can ease this burden by starting to send the frames in each stream as close as possible. After a connection has been accepted, some data specific to the particular media must be loaded into memory. For example, each MPEG video file stored on disk has an associated file containing information about the characteristics of the video, such as size, byte offset in the video file, the frame type and frame number of each frame. This file must be parsed and stored in memory for quick access to frame information.[6] In contrast, GSM audio contains constant sized frames, so very little parsing is necessary. Therefore, it is necessary to delay the start of the audio stream until the video stream is ready to start.

The connections in every session contain unique identifiers. The identifier is used to match related audio and video stream so that simultaneous connections from different clients will not interfere. The stream synchronizer uses a hash table to match up incoming connection requests. The client will send the number of streams that will be created in the session. If more that one connection will be associated with the session, the first connection will be entered into the hash table, will parse any necessary data, and will wait for the other stream. When the other stream is ready, it will notify the waiting connection to proceed and the entry will be removed from the table.

5.2.3 Server Side Engine (`ServerMediaManager`)

Similar to the client side, the server side contains an engine that delegates the tasks needed for the protocol to other objects. The main task of the server side engine is to send data packets to the client at regular intervals. It is also responsible for responding to requests from the client, such as dropping and adding frames to the stream.

5.2.4 Packaging of Data (`FormatPacker`)

Frames of data must be packaged into datagrams to be sent to the client. Recall the packet structure from section 5.1.1. The packet header must be filled and the payload added. Depending on the type of media (audio or video), different objects must be invoked in order to fill in the data properly. The object must first parse apart the necessary files (see section 5.2.2). The object can then respond to requests to create datagrams containing frames of data.

5.2.5 Dropping of Frames

When a request is received by the server to drop frames from the stream, it must decide what frames are to be dropped. For MPEG video, one possibility would be to start at the end of a GOP and start dropping one frame at a time. Each drop request would result in another frame being dropped from the GOP. This approach has two disadvantages. First, since P and B frames are usually much smaller than I frames, little relief would be experienced. Second, since the client is expecting frames at a regular rate, something must be put in the place of the dropped frame when the client decoder requests it. Since P and B frames require references to neighboring frames, anything used to replace the missing frame would result in visual defects. When the first drop

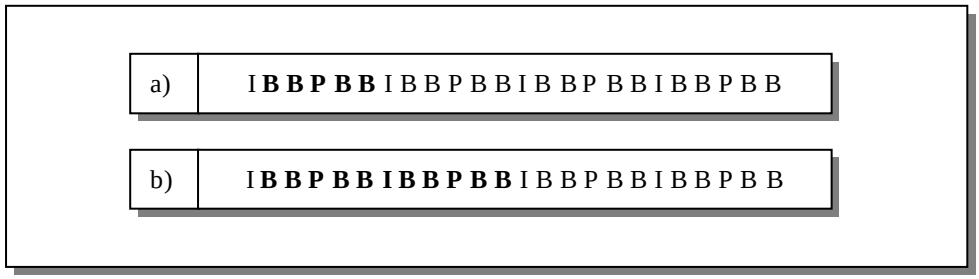

Figure 6 An example GOP sequence. Bold typed letters are those frames that are dropped for a) one drop request and b) two drop requests.

request arrives after a period of no congestion, the frames up to the next I frame are dropped.[7] However, the protocol drops every frame in a GOP, except the I frame, after that. A second drop request would result in dropping all frames in two GOPs except the first I frame. This method continues for each successive drop request from the client (see Figure 6). This minimized the number of dependencies that are broken and thus reduce the visual degradation.

---

[6] This is the Vosaic ".mpg.par" file. The server also uses the Vosaic ".vos" file to obtain information about the difference in start times of video and audio for presentation to the user on the client side.

[7] An alternate method would be to wait until the next I frame to start dropping frames. However, this would decrease response time. For example, for a sequence of 6 fps and a GOP of 12 frames, if the request

Dropped frames are not simply eliminated from the data stream. Instead, packets with no payload are sent. Since the header is only 10 bytes, the overhead is not too great. Packets are still sent so that the client can decide whether a packet was received out-of-order. Otherwise, it would not know whether the network dropped the packet and it therefore must request that the packet be retransmitted, or whether the packet was received out-of-order.

5.2.6 Retransmission of Frames (`PacketHistory`)

When the network drops frames, the client may request the frames to be retransmitted if time is available. Frames are not buffered in memory; rather, if a frame is to be retransmitted, it is simply read from the video file again. One complication that arises is when the network drops a frame that was originally dropped by the protocol (only the header is sent). The client will request the frame to be resent. However, this frame is not really needed for playback, since it contains no data. Therefore, the server need not retransmit the frame. The server needs to keep some sort of record as to which frames were originally dropped by the protocol. This record is maintained in a bit vector. The status of the last x packets sent is maintained, where 1 signifies the packet was sent, 0 signifies the packet was dropped by the protocol (x = 128 for the implementation). Using this simple mechanism, the server can decide whether to send packets that are requested for retransmission.

5.3 Player Framework

The protocol described above is independent of the playback mechanism. Any software that supports the decoding and playback of the media forms can be used. In order to evaluate the resulting quality of audio and video provided by the protocol, several players were developed. Currently, JMF does not support all of the necessary requirements needed for the implementation and was not incorporated into the system. Separate audio and video playback software was developed for the evaluation. Figure 7 illustrates the general design of the audio (GSM) and video (MPEG) players used in the implementation. Frames are read from the *Protocol Buffers* and placed into a contiguous circular *Shared Buffer*. This buffer is constantly kept full by a thread (`BufferLoader`) that loads frames into the buffer. The *Media Player* is then free to read from the *Shared Buffer* and can present the data to the user using the *Playback Device*. New players for

---

to drop arrived when the server had just sent an I frame, it would have to wait almost 2 seconds until it could start dropping frames.

different media formats can easily be incorporated into the player framework. The specific parts of each player are described below.

5.3.1 GSM Player (`GSMPlayer`)

The audio decoding software used was developed by Vosaic Inc. and is part of the TVStation playback software [11]. It converts GSM compressed audio to μ-law format. The conversion process is done entirely in Java. Data is sent over the network in GSM format and is converted before the data is sent to the audio device using the `sun.audio` package (this package will be superceded by JMF). GSM has higher compression ratios than μ-law, so the extra process of converting the data is acceptable, because the transmission of the data takes less bandwidth and is less likely to cause congestion. The `sun.audio.AudioPlayer` class reads data from a stream and sends the data to the native audio device. The player cannot read directly from the protocol buffers, since the data must be first converted. The small intermediate shared buffer is used to hold the μ-law format data from which the player reads; the GSM data is converted before it is placed into the shared buffer.

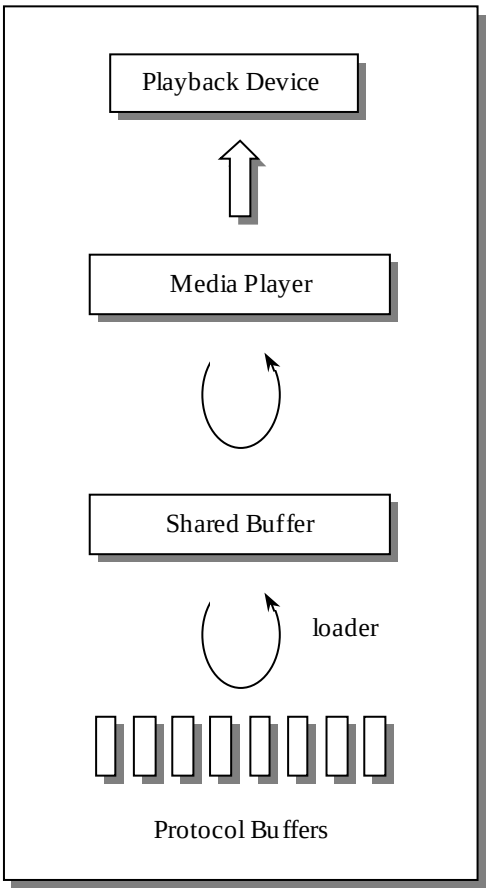

Figure 7 Design of media players. The loader places data into the circular shared buffer. The media player reads data at a controlled rate and sends it to the playback device.

5.3.2 MPEG Player (`MPEGPlayer`)

The MPEG video player follows the same design as the audio player. The MPEG decoder is the only component that requires native code. A modified version of the mpeg2play software developed at the University of California at Berkeley was used as the MPEG decoder. A Java interface was placed on top of the native code to allow access from the Java classes. The decoder was modified to make it flexible enough to read from any source (i.e., file, network, buffers, etc.). [8]

[8] Any class that extends the `java.io.InputStream` abstract class may be used as the data source.

The media player requests frames at regular intervals (the period of the video sequence) and presents the images (`VideoWindow`).

5.3.3 Media Player (`MSPStreamPlayer`)

The players were developed to present the different media formats. A player that presented both audio and video simultaneously was developed to create a combined format session. The combined player simply creates individual players (`BufferedMPEGPlayer` and `BufferedGSMPlayer`) and starts them together. The only complication is synchronization of the streams.

5.3.4 Synchronization of Streams (`StreamSynchronizer`)

After the creation of the players, data packets begin being sent to the client (recall each media stream uses a different connection). The streams must be started at the correct time to achieve proper synchronization. The same synchronization mechanism described in section 5.2.2 was used. Before the players may start, they must prefetch some packets into the shared buffer. After both players have buffered the correct amount of data, they are allowed to proceed.

The synchronization of streams is achieved by running each stream against an absolute clock (see section 5.1.2). The rates of both streams are known. Therefore, synchronization is required only at the start of the streams. After the start of the media playback, no further synchronization is needed.

6 Simulated Network (Network)

In order to test the protocol, media must be sent over a network and congestion must be present in the network. For congestion to appear, the capacity of the network must be low enough for

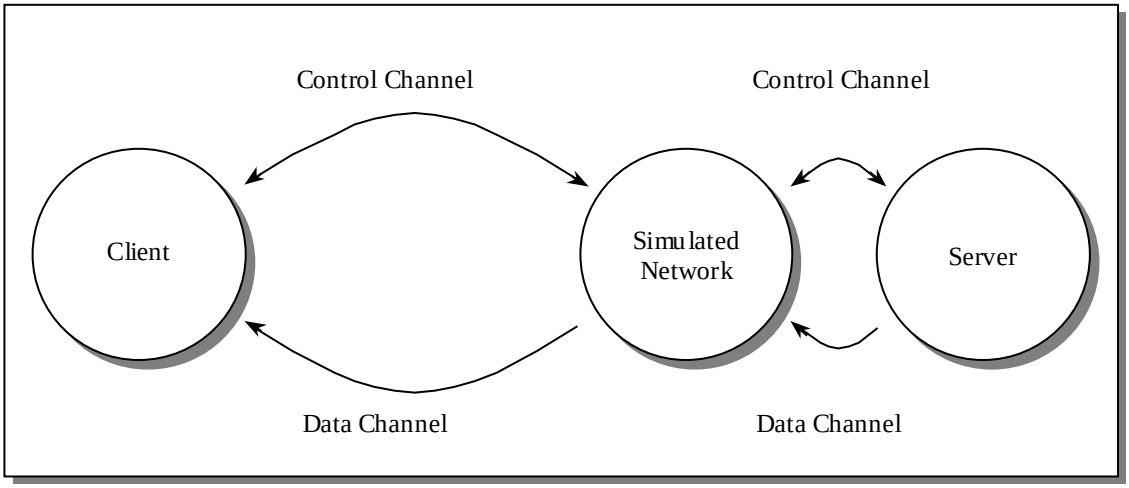

Figure 8 Diagram of simulated network. A component is placed between the client and server to delay data packets. The baud rate at which packets are sent may be adjusted. Low baud rates may result in buffer overflow and loss of packets.

frames to be dropped and delayed. For a given network, the amount of data transmitted must approach the limit of available bandwidth. This requires that the video sequences are encoded in such a way that this limit is reached (audio is encoded as CBR, so no adjustments may be made). [9]

An environment was created to simulate the congestion on a network. The network bandwidth could be tuned to different baud rates to test varying amounts of congestion. The simulated network also allowed experiments to be more accurately reproduced, as well as allowed experiments to be conducted on a single machine, eliminating the need for a real network connection.

The network simulation component was introduced between the client and server. All data flowed through this component. This component had the ability to modify the normal flow of data between the client and server.

[9] Encoding at a higher rate, larger frame dimensions, or more bits per pixel, may increase the size of the video sequence.

6.1 Design

Two approaches were considered in the design of the simulated network. The first design provided three variables to alter network characteristics:

1. Percentage of dropped packets.
2. Burstiness.
3. Number of out-of-order packets.

This design tested certain aspects of the protocol, but did not simulate a real network very well. The second design did a much better job of simulating a real network.

The second design tried to reproduce more accurately what was happening within the operating system with respect to the network connections. The simulator focused on providing congestion on the data (UDP) channel. The control (TCP) channel was not subject to congestion (messages sent via the control channel were never delayed). Allowing the control channel to become congested would have made the simulation significantly more complex. This had a small effect on the overall simulation, since most of the data is transferred via the data channel. Neglecting congestion in the control channel would give a slightly better response to the simulated network. However, the main goal of the simulation was to approximate some of the trends that would occur in a real network.

Recall the discussion in section 3.3. Point-to-point network connections consist of two endpoints, possibly with several routers in between. Each router accepts packets and transfers them to the next stop of the path. The operating system allocates a certain amount of buffer space for each connection. It is possible for the allocated buffers to overflow, if too much data arrives before it can be sent out again. UDP will silently discard the packets that do not fit in the buffers. The network simulator emulates this behavior by delaying the rate at which packets can be sent. By dividing the size of the packet by the baud rate, the time to send the packet can be calculated. The simulated network sends frames and waits for the remaining amount of time before sending the next packet. For example, if a packet is 1000 bits and the baud rate is 2000 bps, the packet is sent and then waits for 0.5 sec (minus the actual time to send the packet). This allows packets to build up in the operating system kernel buffers, which could possibly overflow (i.e., packets would be dropped). The baud rate can be adjusted to vary the speed at which packets can be sent over the (simulated) network, thus varying the amount of congestion experienced.

Since the second approach followed the behavior of a real network more closely, it was adopted for the video transmission experiments.

6.2 Comparison of Real and Simulated Networks

The simulated network was designed to reproduce some of the trends that appear in a real network in a more controllable environment. A comparison was made between the real and simulated networks to determine how accurately the simulation could capture the trends of the real

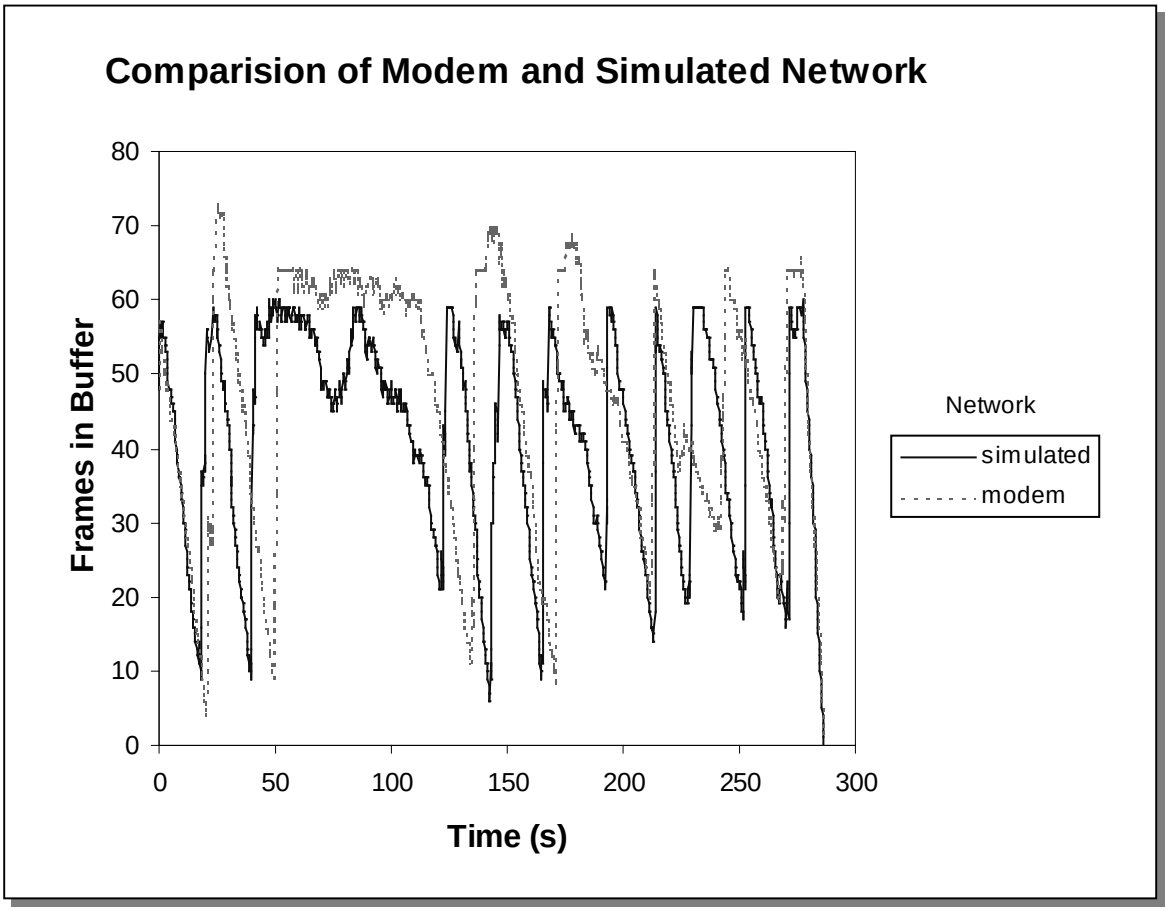

Figure 9 Graph comparing the number of frames in the protocol buffer using the protocol over a modem connection and over the simulated network.

network during a video transmission. The comparison was made by observing the number of data packets present in the protocol buffers with respect to time. The video sequence used in all of the following experiments was an MPEG sequence of George Michael's video "Jesus to a Child". The video sequence was comprised of 1718 frames encoded at 6 frames per second with a GOP

of IBBPBBPBB. The image dimensions were 160 x 120 pixels. The experiment compared the sequence running over a 14.4 baud rate Point-to-Point (PPP) modem connection and the simulated network with baud rate set to 9000 bps. The simulated baud was set lower to compensate for overhead within the modem connection. The PPP connection also traversed four intermediate routers, which may increase the delay of packets.

As can be seen from Figure 9, many of the trends of the real network are captured. At the beginning of the sequence, the size of the buffers decreases rapidly. This rapid decrease occurs because the video has large amounts of motion in the beginning sequences, which creates larger frames. The protocol then reduces the amount of data being transmitted and the buffer size increases. The size remains high between 50 and 150 seconds and then decreases again.

Two observations can be made from the comparison. First, the maximum and minimum buffer size is often greater for the real network. Second, the real network values generally lag behind the simulated network. Both of these observations are due to the fact that the control channel did not introduce delay in the simulated network. By adding delay into the control channel, the real network reduces response time of client messages to the server. If the client sends a message to the server requesting a drop of frames from the stream, the server on the real network gets the message slightly later than the server on the simulated network. This means that the flow of data will not be decreased as quickly and congestion continues for a longer period of time. This decrease in response time forces the size of the protocol buffers to drop further before the decrease in data traffic can take effect. With this decrease in response time, the general rise and fall of the buffer size lags the simulated network. Although the comparison is not exact, many of the important trends are followed and it provides a sound basis for further experimentation.

7 Results

Experiments were performed to gauge the effectiveness of the protocol. The first experiment compared the quality of playback with the protocol operational and with the protocol turned off. That is, packets were buffered and placed in the correct order in the protocol buffers, but no adaptation was attempted. Experiments were also performed to determine the optimal parameters to the protocol.

7.1 Adaptation v. No Adaptation

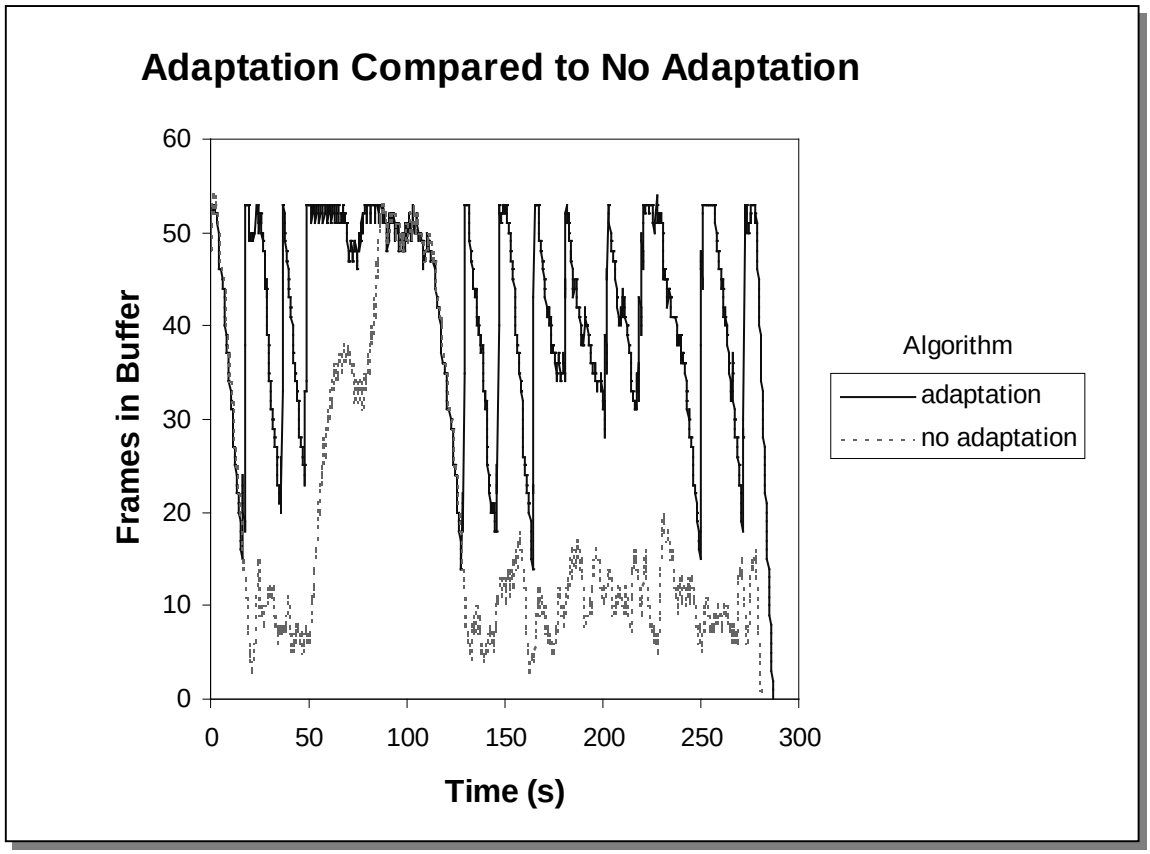

Figure 10 Graph comparing the number of frames in the protocol buffers with the adaptation algorithm turned on and off.

It is interesting to see what effect the adaptation has on the size of the buffers as compared to no adaptation. All aspects of the protocol were identical for the case of no adaptation, except for the fact that no action was taken when the buffer decreased in size and no frames were retransmitted (essentially UDP with reordering of packets). Figure 10 shows the results of this experiment.

The baud rate was set to 10000 bps and the buffer time was 8.0 sec. When the protocol used adaptation, the slot time was set to 1.0 sec and the check time to 6.0 sec. It can be seen that the case of no adaptation gave a substantially lower number of frames in the buffer for every point during the video sequence. Although the buffer never did underflow, it was at risk of doing so for almost the whole sequence. Any slight increase in congestion at any time would have caused the buffer to underflow, leaving no data in the buffer.

Let the *vulnerable period* be defined as the period during which there are only ¼ of the desired

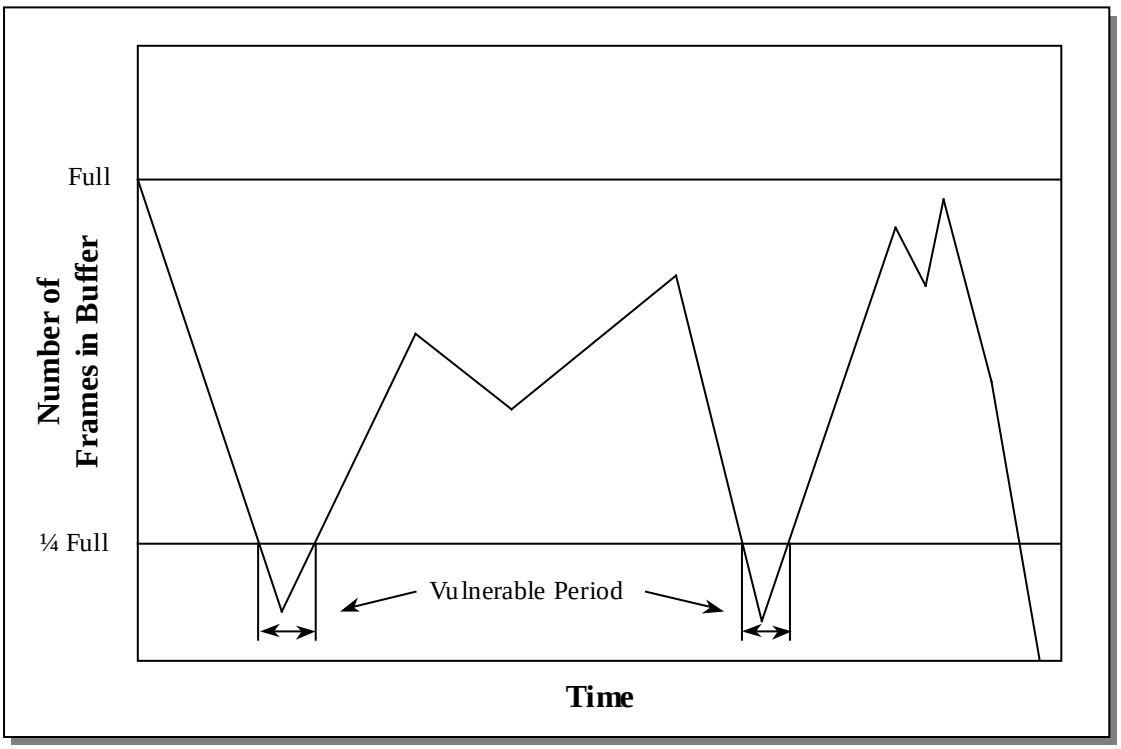

Figure 11 Explanation of vulnerable period. The buffer enters the vulnerable period when the number of frames is ¼ of the full buffer.

number of frames. This value can represent the period in which the buffer is close to becoming empty. It can be seen that for this sequence, the buffer is in the vulnerable period for much of the sequence.

There is a further problem that is not visible from the graph; the quality of the video deteriorates when there is no adaptation. The problem is that the network drops packets randomly, while the adaptive protocol drops less important frames (P and B). When frames are dropped at random, it

is equally likely to drop P and B frames as it is I frames. I frames are the most important frames, since they are references for other frames. If an I frame is dropped, every frame in the GOP cannot be rendered correctly. If only P and B frames are dropped, which is what the protocol does, the video may pause on a particular frame for more than the normal amount of time, but there will be no visual defects in the displayed images. This is important for the acceptability of the video quality for the user.

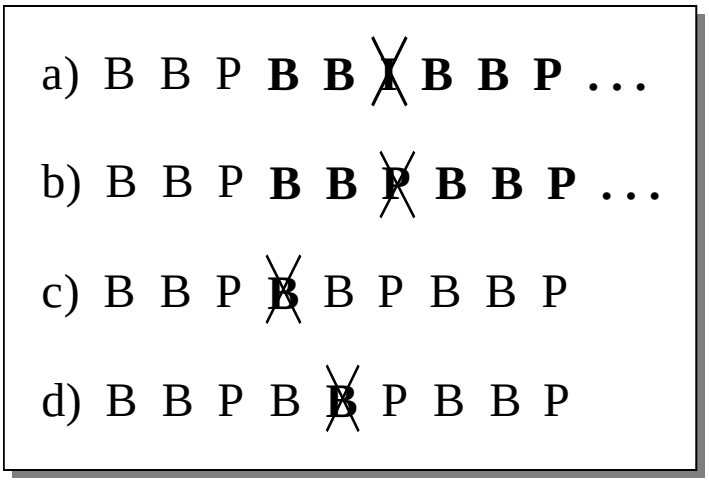

Figure 12 Illustration of frame dependencies that are broken if frames are missing. Bold characters represent frames will be rendered incorrectly if the frame crossed out is missing. The "…" signifies that the remaining frames in the GOP will be incorrectly decoded. A missing I frame (a) will cause the previous two B frames and all remaining frames in the GOP to be incorrect. A missing P frame (b) will cause the previous two B frames and all remaining frames in the GOP to be incorrect. If the first of two consecutive B frames is missing (c), it will be incorrect. Similarly, if the second B is missing (d), it will be incorrect as well.

7.2 Quality of Video

A further study was conducted to measure the quality of the video playback for the algorithm with and without adaptation. Quality was measured as the number frames that were correctly displayed. Incorrectly presented frames may result from a broken dependency or an I frame that is presented more than once in place of another frame (dropped by the protocol). A missing frame will result in frames being decoded with incorrect information (differences are being taken from the wrong frame). A broken dependency will result in degraded quality, since the frame is being decoded incorrectly, using an incorrect reference frame. For example, for the GOP sequence described above, a missing P frame will result in all frames after it in the GOP to be incorrectly decoded, as well as the previous two B frames. Therefore, for this particular GOP sequence, if the first P frame in the GOP is missing, there will be 11 broken dependencies. Figure 12 describes the broken dependencies for the four different possible scenarios.

Figure 13 shows the results of this study. It can be seen that for large amounts of congestion (low baud rates), the adaptation algorithm was able to reduce the number of incorrectly displayed

frames. Recall that most of the frames that are dropped by the protocol are a full sequence in the GOP (except the I frame). Although, the video may appear to pause, there is a minimum of

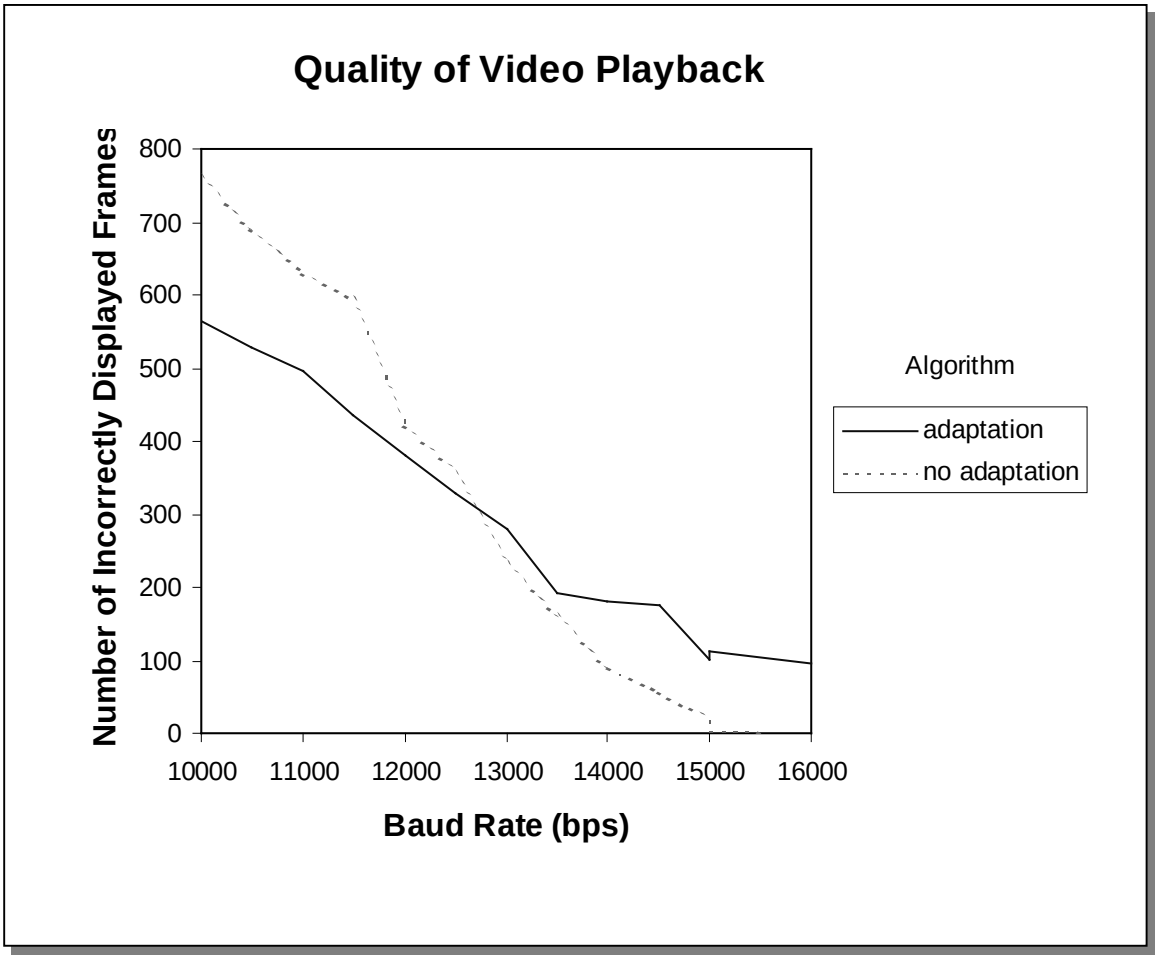

Figure 13 Graph of video quality with the adaptation algorithm on and off. Video quality is measured as the number of frames that are correctly displayed.

"blocking" effect, which occurs when random frames in the sequence are missing. The data for the algorithm with adaptation includes frames that were dropped by the protocol, which are not actual broken dependencies, since they merely pause the video. However, these frames may be considered a factor in the measurement of quality, although they are not as strong a factor as actual broken dependencies.

It can also be seen that, for low levels of congestion, no adaptation produces less incorrect frames. This occurs because the protocol reacts to the reduction in data flow before the network drops frames. The amount of data is purposely reduced in order to save bandwidth in anticipation

of further congestion. However, in this case, the congestion does not occur and the algorithm is slightly ambitious.

Figure 14 shows the actual broken dependencies. It can be seen that the number is small. For example, at a baud rate of 12000, the number frames that are incorrectly displayed is 381. However, only 10 of those were a result of incorrect dependencies.

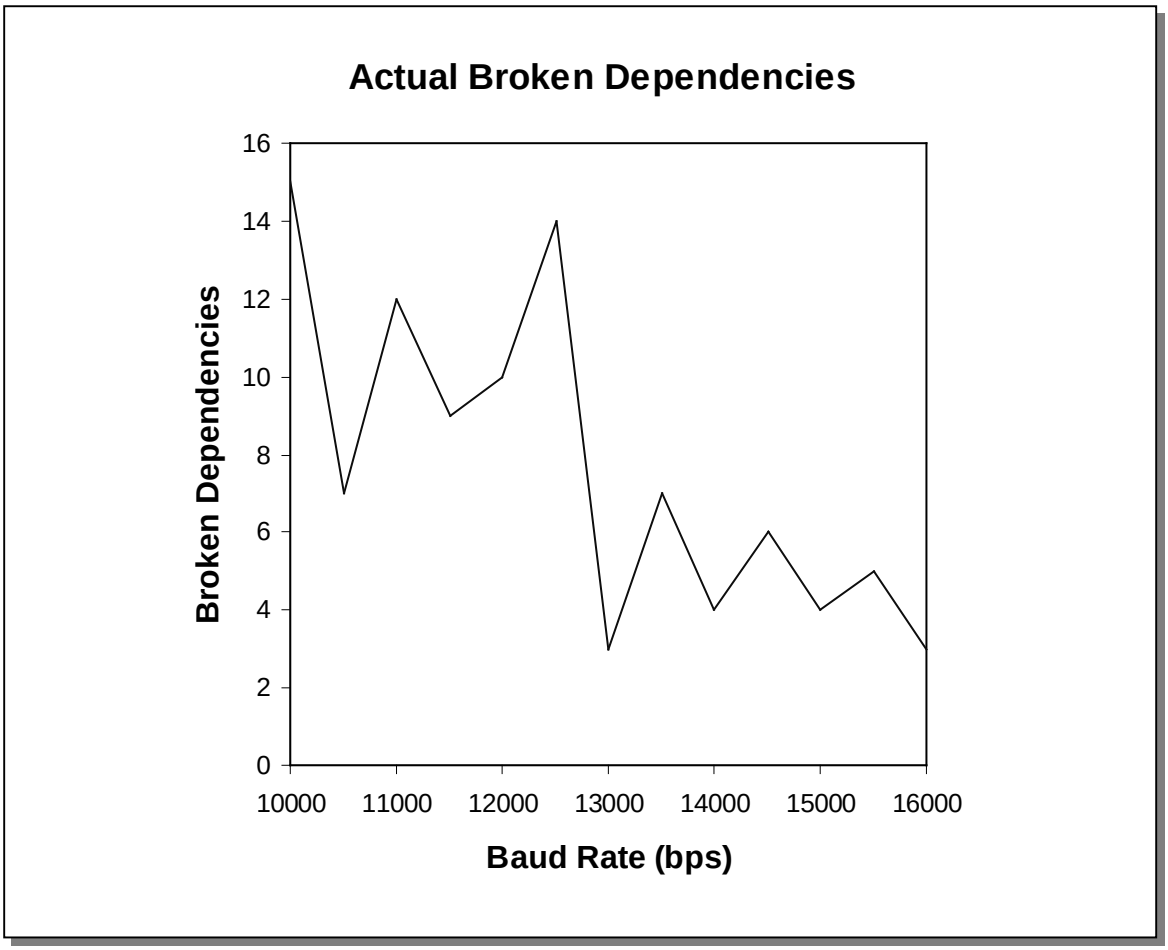

Figure 14 Graph showing the number of actual broken dependencies when the adaptation algorithm is operational.

7.3 Varying Check Time and Slot Time

The following experiments were performed to find the values of the parameters to the protocol (buffer time, check time and slot time) that produced the best results. Although the parameters

will give slightly different results for different video sequences, they provide a general measure of good values for the parameters.

The check time and slot time where varied to investigate the effects on number of missed and dropped frames. Missed frames are frames that were never received by the client (dropped by the network) and dropped frames are frames dropped by the protocol (only packet headers arrived with no payload). Video sequences were tested using three different slot times (0.5, 1.0 and 1.5 sec) and the number of missed or dropped packets were recorded for different check times (4.0 to 7.0 sec). Each was run with a buffer time of 8.0 sec and a network baud rate of 10000 bps.

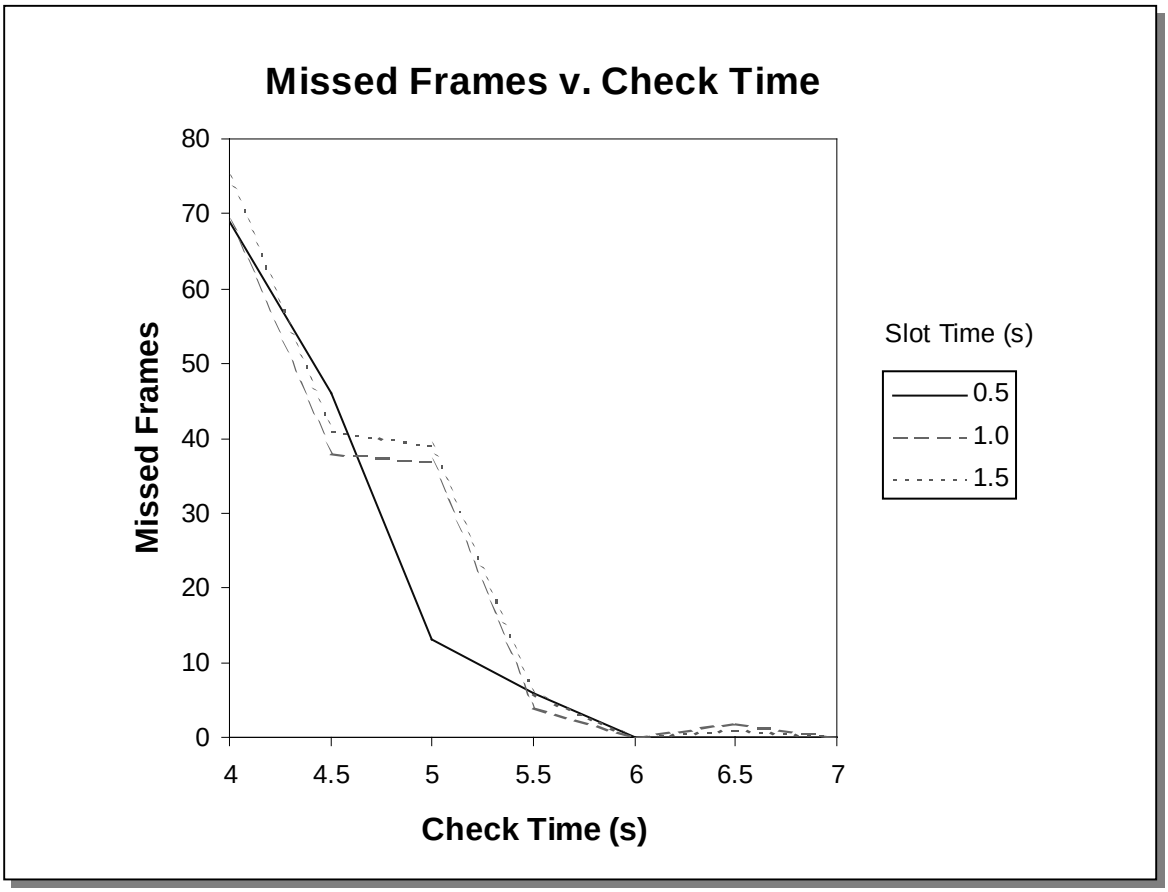

Figure 15 Graph showing the number of missed frames with varying check time and slot time.

7.3.1 Missed Frames

Figure 15 shows the results of missed frames. It can be seen that the number of missed frames decreases for increase in check time for every slot time. When the check time is high, the protocol can start to drop frames before the network does. As the check time is decreased, the protocol

does not react fast enough and the network starts to drop packets. In general, the slot time of 1.5 sec had a higher number of dropped frames. When the slot time is too large, the protocol cannot react quickly enough to the decrease in buffer size. Slot times of 0.5 and 1.0 sec produced mixed results, which is partially do to the characteristics of the particular video sequence.

7.3.2 Dropped Frames

Figure 16 shows the number of frames dropped by the protocol as check time is varied for the same three slot times. The general trend is for the number of dropped frames to increase as the check time is increased. This seems an obvious result of requesting that frames be dropped sooner. When the check time is relatively large, smaller decreases in the size of the buffer will

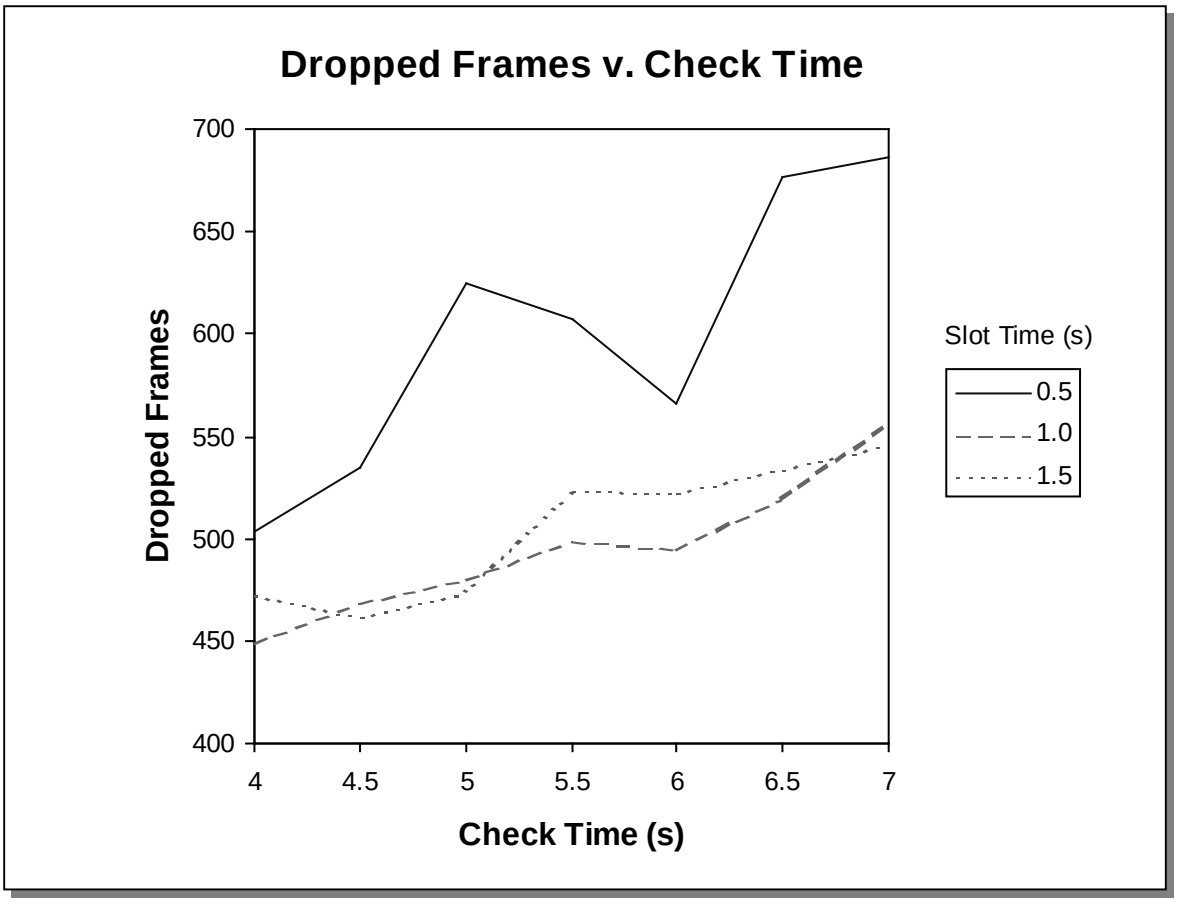

Figure 16 Graph showing the number of dropped frames with varying check time and slot time.

trigger the sending of messages for the server to start dropping frames. The chart also shows that the slot time of 0.5 sec generally produces the most dropped frames, with slot times of 1.0 and 1.5 sec producing similar results to each other. This follows the same reasoning as above. With the

slot time relatively small, the protocol becomes very sensitive to changes in the buffer size. Once the first drop request is sent to the server, small decreases in the size of the buffer trigger further requests.

7.4 Wait for Processed Requests

One further study was performed to answer the question of whether the protocol should respond to every change in the current slot, or should wait for previous messages to be processed before issuing new messages to drop or add frames to the stream. For example, if a drop message is sent to the server and the size of the buffer drops enough to send another drop message before the effects of the first is observed, should the second message be sent? The algorithm was altered to only allow drop and add messages to be sent to the server if previous such messages had been processed by the server and the client was observing the results. This was achieved with the *drops* field in the packet header. The client maintained a record of how many drop requests were currently issued. Every time a drop message was issued, the record was increased by one and decreased by one for each add message. The server also maintained a similar value and placed it in the *drops* header field. The client could not issue any further messages until its current record matched the value in the header. The idea was to give the server some time to react to any changes it must perform.

It was found that by *not* waiting for the server to respond to drop and add messages, better results were achieved. If the client waited for a drop or add message to process before sending the next message, it took too long to get the confirmation in the header if many small frames were backed up in the UDP receive buffers. Since the protocol could responding quickly enough, the buffer often drop lower. When the request did finally get processed, the buffer may have decreased passed several slots and the next message may request several drops at once. This lead to decreased smoothness in the flow of data to the client.

7.5 Multiple Streams

The previous experiments were performed to evaluate the effectiveness of the protocol for the delivery of video. However, most sessions will include both video and audio. Therefore, it is most interesting to observe how the protocol performs when both audio and video streams are being transmitted. As described in section 3.4, audio and video must be treated differently due to the difference in human perception of various media types. It is necessary to drop video before audio to produce a minimum of annoyance to the user. When congestion occurs, the dropping of

video should relieve congestion so that the audio stream may be preserved. Only if congestion is so great that dropping a significant number of video frames does fully relieve the condition should audio be dropped as well.

This experiment used the same MPEG video sequence as was used above, with the addition of the GSM audio stream. The two streams were sent over a PPP modem link at 28.8 baud rate. (The simulated network was not used because accurately modeling the interaction of multiple streams

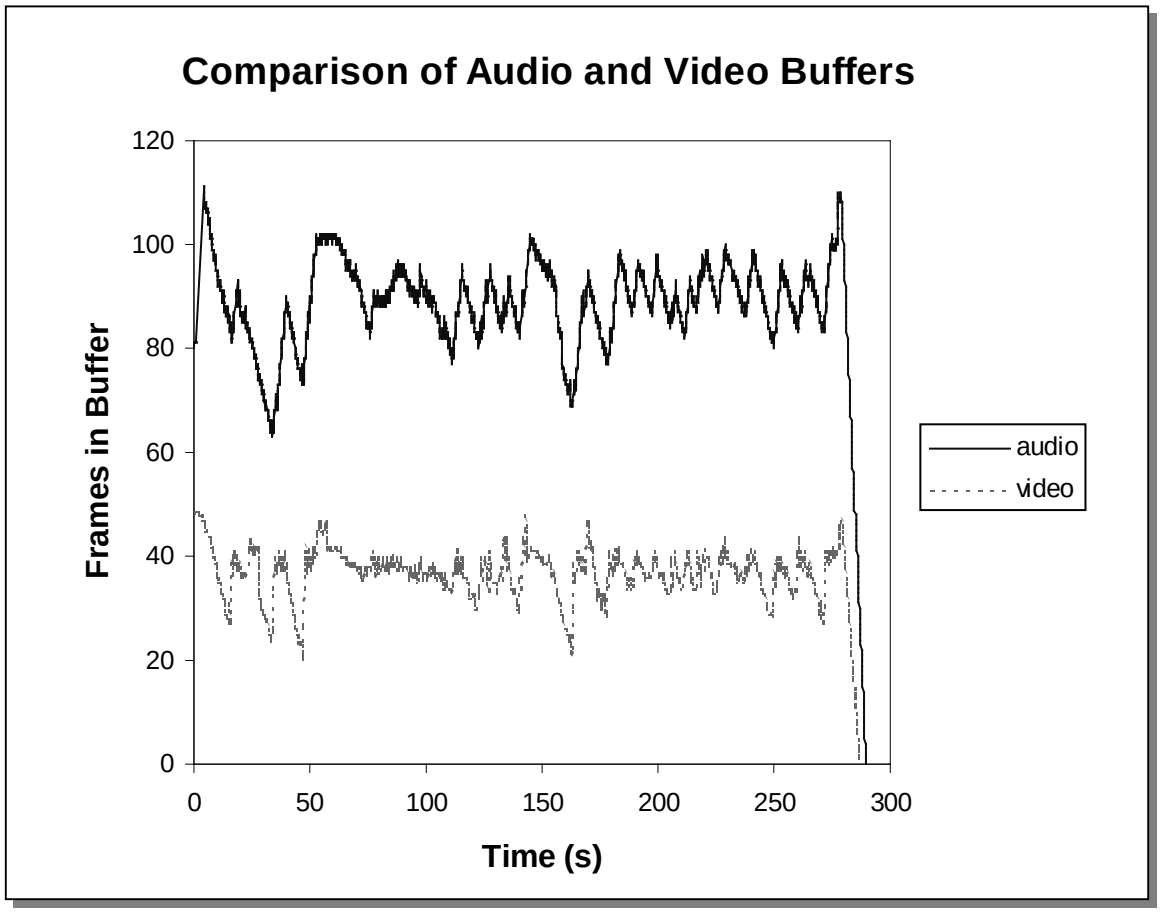

Figure 17 Graph comparing the number of frames for video and audio streams. When the number of frames in the buffer drops due to congestion, the dropping of video frames allows both streams to recover and eliminates the need for audio frames to be dropped.

within the operating system becomes complex.) An audio packet was sent every 100 ms, which contained 165 bytes of data. Figure 17 shows the results of the experiment. The number of frames in the protocol buffers is shown with respect to time for both audio and video. The buffer time was set at 8.0 sec. The video check time was 6.0 sec and the audio check time was 3.0 sec. The slot time was 1.0 sec for each of the streams.

For the video stream, the protocol dropped 640 frames and the network dropped 1 frame. For audio, however, neither the protocol or network dropped any frames. It can be seen that when the number of frames in the video buffer decreased, the audio buffer decreased also. The dropping of video frames was enough to relieve the congestion and allowed both the video and audio buffers to increase again. The number of audio frames never dropped so low that it was required that they be dropped also.

7.6 Discussion

There is a trade-off when deciding what values to use for the check time and slot time. If the check time is high and the slot time is low, then more requests to drop frames are sent sooner (the buffer hasn't decreased in size that much) and the network is not given time to possibly clear itself of congestion. By letting the protocol drop frames earlier, we purposely eliminate frames from the stream, which will cause the video player to have the illusion of pausing between (I) frames. On the other hand, if the slot time is larger and we let the number of frames in the buffer drop too low, the playback quality may remain high for slightly longer, but the network will eventually drop the packets at random and the display will be degraded. Although both options increase the number of dropped frames, the protocol drops the frames in the former and the network drops the frames in the latter. When viewing the video sequence in both cases, having the video pause is better than letting the video quality degrade. When playback is degraded, the images can become very "blocky" and it is difficult to observe the images clearly. The protocol has knowledge of which frames are more important and can make intelligent decisions as to what frames can be eliminated from the stream with the least amount of visual defects.

8 Future Work

The current implementation of the protocol was written in Java. However, the MPEG decoder was written in native code. This makes the inclusion of the system into HTML web pages more complicated, since the web browser cannot transmit the native code. JMF will allow the playback of audio and video sequences and will be incorporated into future versions of web browsers. Currently, JMF does not support all the necessary functionality on the two major platforms, UNIX and Windows NT. Once JMF includes this support, it would be desirable to use these audio and video players with the protocol. This would eliminate the need for native code and would greatly simplify the distribution of the protocol for use in web pages.

Future work should also consider smoothing the video to create less bursty traffic. The transmission of the video could be smoothed over a number of frames. The protocol would have to be adjusted if this addition were to be added. The protocol operates by observing the fluctuations in the number of frames in the buffer. Every time an application reads data from the buffers, the determination is made as to the current state of the system. By smoothing the video, frames would not be sent at regular intervals and the buffer size would begin to vary, even if congestion was not present. However, if the check of the buffers was made only after the number of frames that were to be smoothed was sent, the size of the buffer should be the same compared to no smoothing. This may reduce the response time of the protocol, since the network may actually be congested, but the protocol may not be able to observe the condition until the smoothed frames have been transmitted. Therefore, the best number of frames to smooth would have to be determined. However, this may lead to better allocation of bandwidth in the network and may result in less congestion.

9 Conclusions

The excessive amount of time required to download audio and video files has made video streaming a popular alternative to reduce video startup latency. Both lossless and lossy techniques have been developed to stream video. In this thesis, a new lossy technique was developed, which delivers data in real time and adapts to problems in the network and at the server. The protocol attempts to be fair in its consumption of network bandwidth. The implementation used MPEG video and GSM audio as the forms of media. However, the protocol is not limited to these media forms; different media types may be added to the framework to allow the streaming of a diverse range of media.

Ease of distribution is a key factor in acceptability of such a protocol. With this goal in mind, the implementation was written in Java. However, the MPEG decoding engine required native methods to provide the speed necessary to decode frames. The inclusion of JMF in future versions of web browsers will eliminate the need for native code and will enable transparent distribution of the protocol. Therefore, future work should include using the JMF software decoders with the protocol.

Studies were performed to evaluate the effectiveness of the protocol. It was found that the protocol reduced the number of visual defects when compared to the system running without the adaptation algorithm operational. When the algorithm was turned off, the network dropped packets at random, causing a reduction in video quality. The algorithm was able to make more intelligent decisions as to which frames should be dropped when congestion occurred. Also, with the adaptation algorithm off, the number of frames in the protocol buffers remained low for almost the entire video sequence. The buffers run the risk of underflowing in this case.

Experiments were also performed to identify reasonable values for the parameters used in the protocol. The tests showed that buffer time = 8.0 sec, check time = 6.0 sec and slot time = 1.0 sec produced the best overall results for the particular video sequence used. Although these values are somewhat specific to the characteristics of the particular video sequence, they provide approximate values for video sequences of this frame rate over a modem. Video sequences travelling over higher bandwidth connection may need to adjust the values. For example, the buffer time could be reduced, since congestion may be less likely to occur, thereby providing less latency at startup.

When using the protocol for both video and audio streams, the dropping of video frames was found to relieve congestion so that dropping audio frames could be avoided. The check time of audio was set lower (3.0 sec) than the video check time (8.0 sec) to reduce the chance that audio would be affected. This is necessary due to the difference in sensitivity of the human senses.

It is critical that the continuous use of network resources required by streamed media not overload the capacity of the Internet. The type of adaptive protocol described in this thesis will become increasingly important as more network bandwidth is consumed by the transmission of audio and video data.

Appendix A

Class Diagrams

This appendix gives examples of some of the class diagrams in the Unified Modeling Language (UML). The diagrams are presented to make the design of the system clearer and may also be used as a guide when looking at the Java source code.

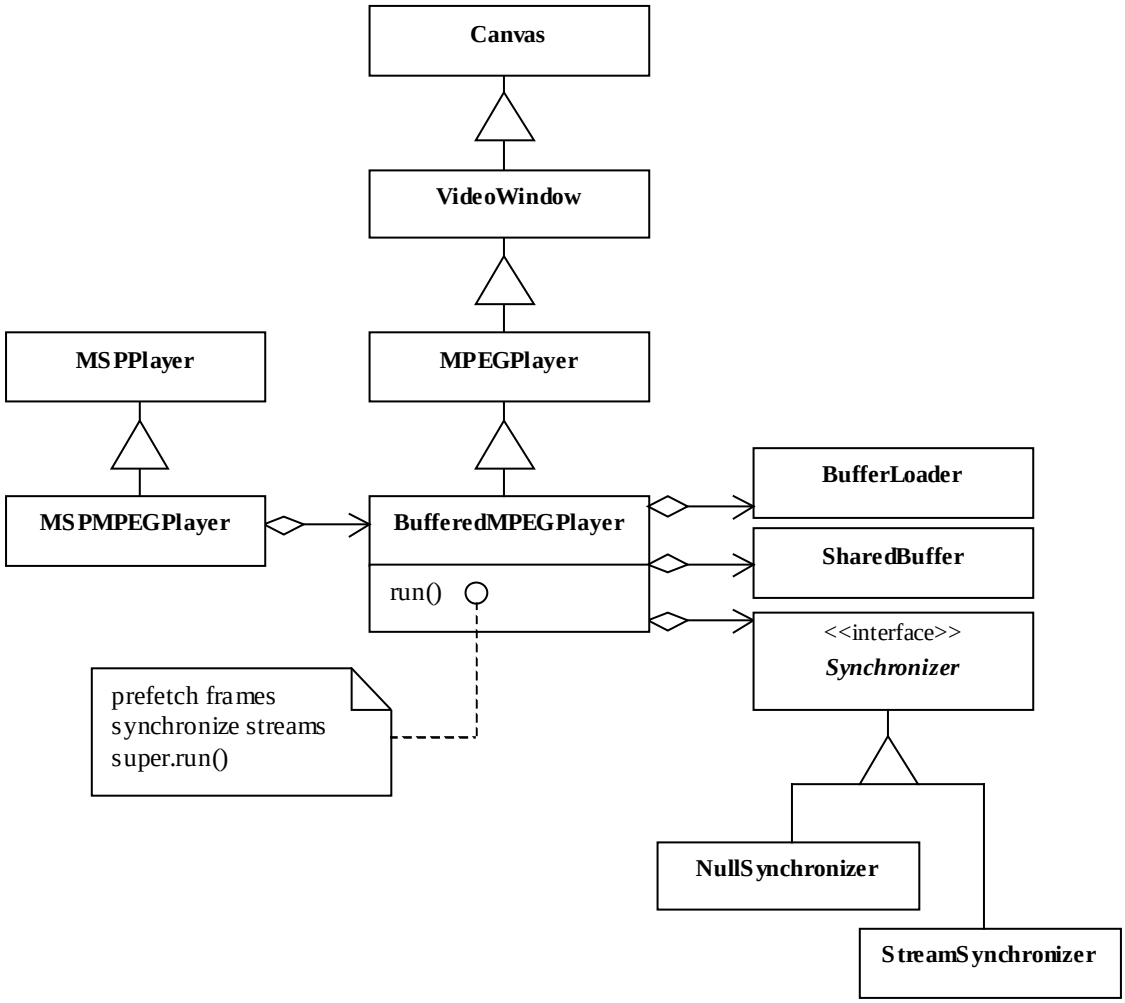

Figure 15 UML diagram of MPEG video player. Any input source may be used as a source to the BufferedMPEGPlayer. The input source is used by the BufferLoader to load data into the SharedBuffer. The MPEGPlayer then reads frames at a given rate and displays them in the VideoWindow. The MSPMPEGPlayer provides the data source from the MSP protocol stream.

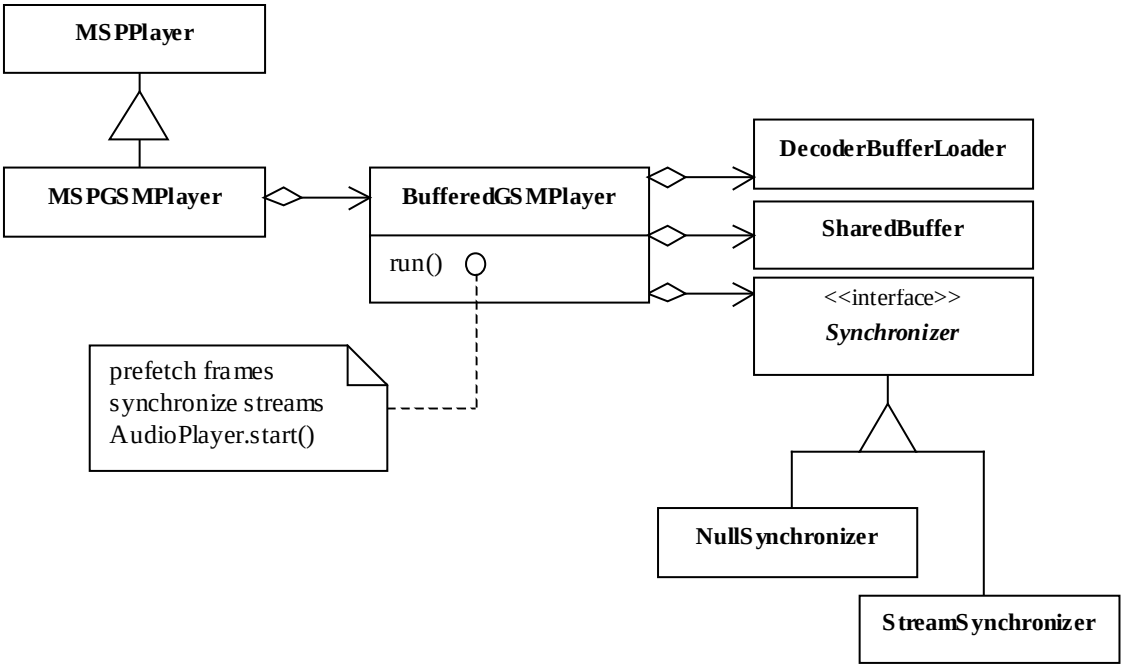

Figure 16 UML diagram of GSM audio player. Any input source may be used as a source to the BufferedGSMPlayer. The input source is used by the DecoderBufferLoader to decode GSM format audio data to μ-law format and load the data into the SharedBuffer. The MSPGSMPlayer provides the data source from the MSP protocol stream.

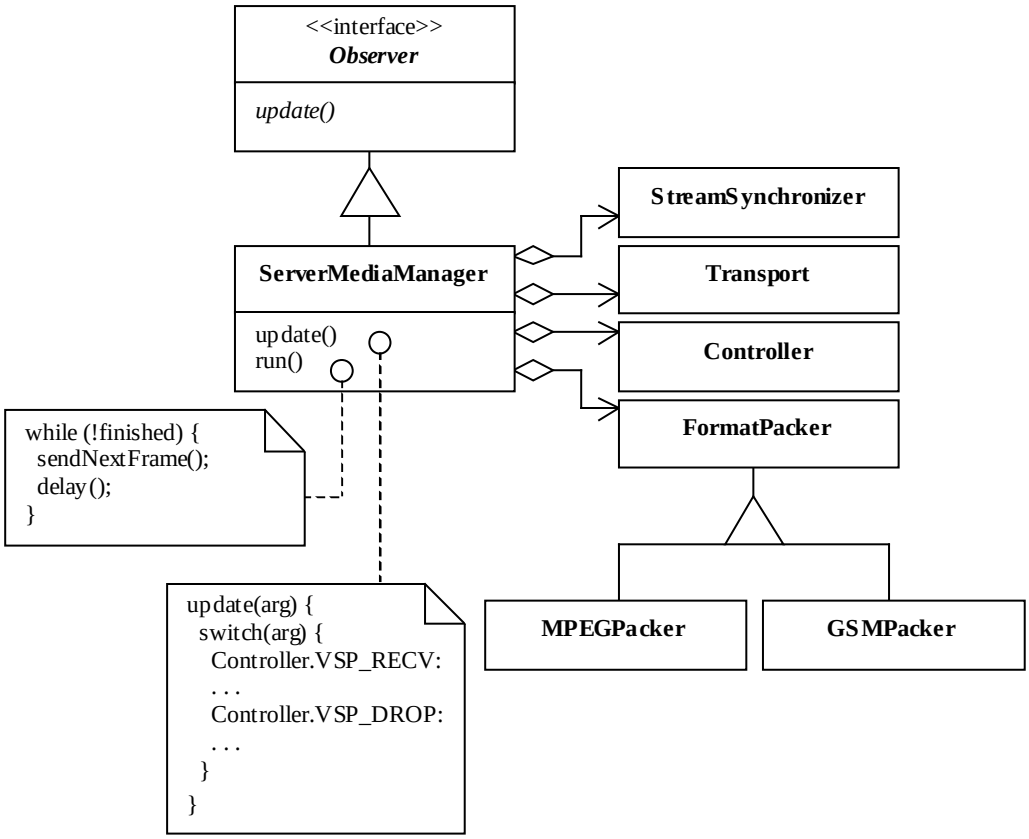

Figure 17 UML diagram of the server side. The ServerMediaManager class is notified of events from the Controller. The FormatPacker creates the datagram packets and they are sent via the Transport class. If the session includes both audio and video streams, the StreamSynchronizer will start sending of the two streams at the same time.

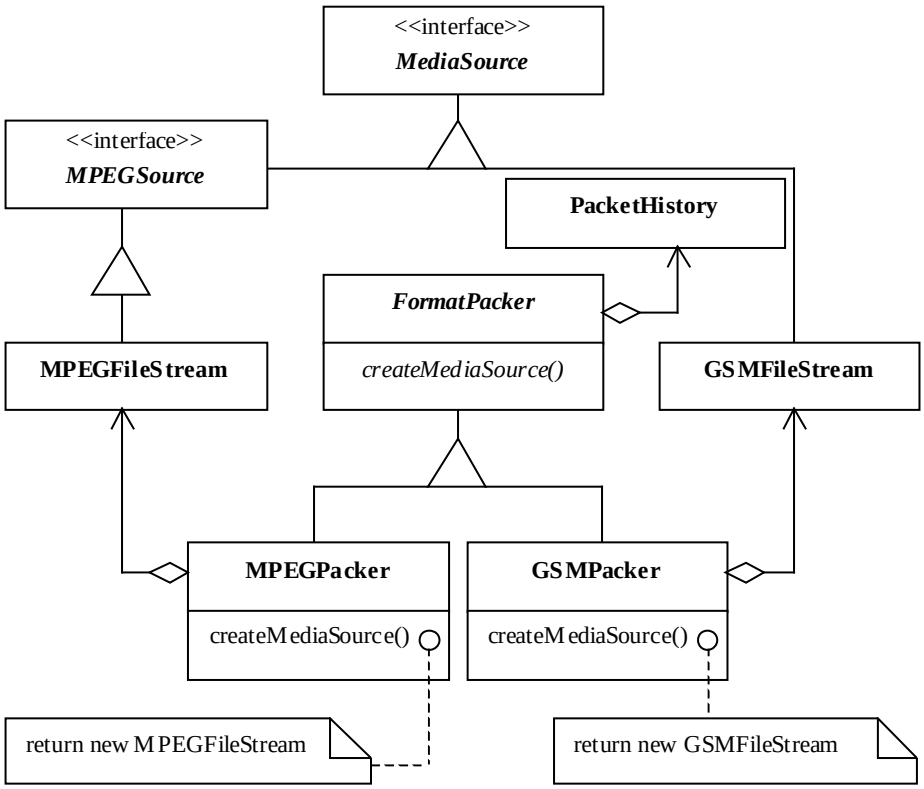

Figure 18 UML diagram of specific packer classes for audio and video. New classes may be added to the framework to read from sources other than files.

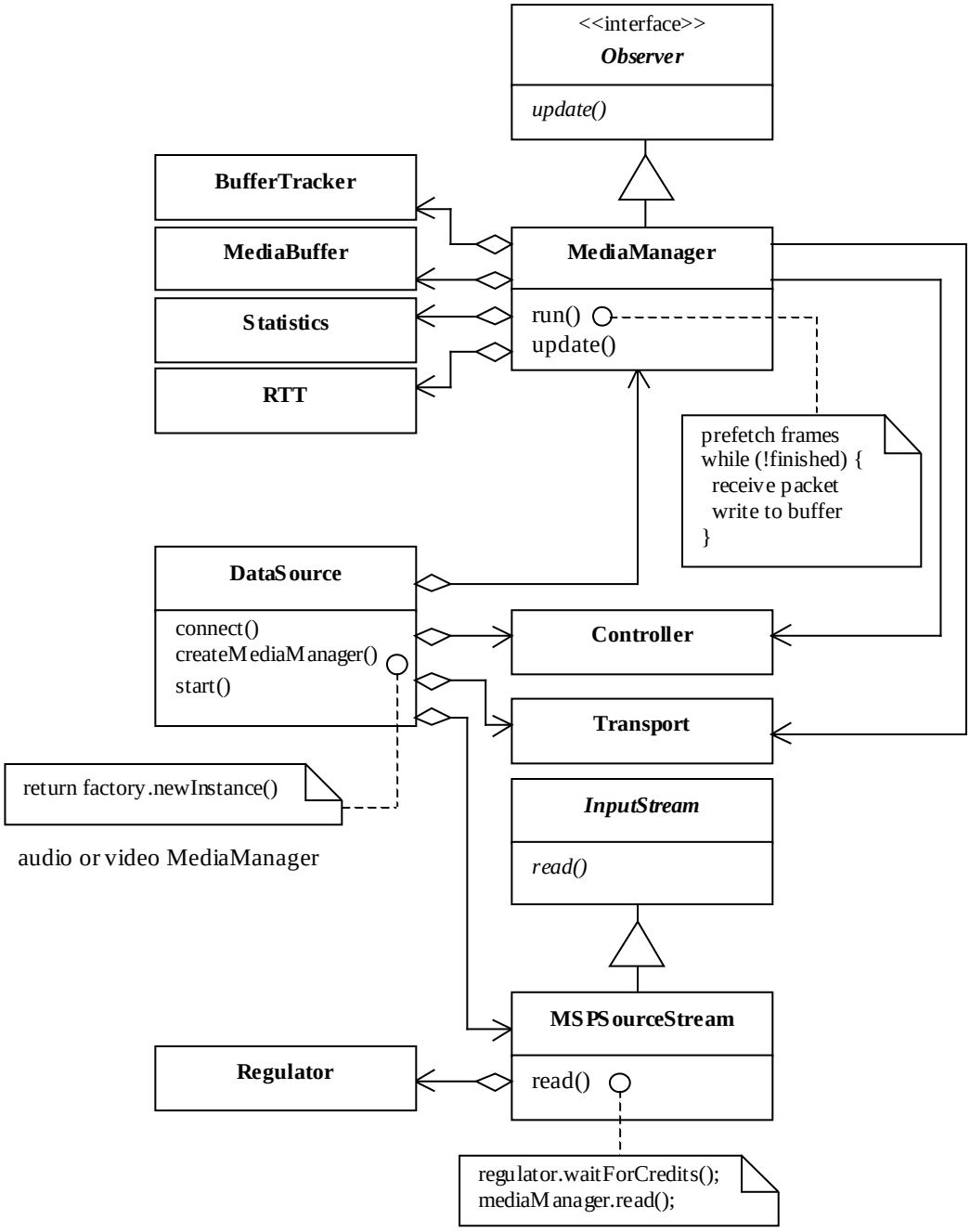

Figure 19 UML diagram of client side. The DataSource creates the connection to the server through the MSPSourceStream. Both the control (Controller) and data (Transport) sockets are created. The appropriate MediaManager (audio or video, depending on what factory is installed) is created. After the handshake, the data starts to be sent by the server. The MediaManager delegates to the BufferTracker, MediaBuffer and Statistics for critical operations of the protocol. It also gets notified when the Controller receives round trip time responses and forwards then to the RTT class. Applications that read from the protocol buffers may only read one frame of data at a time, which is throttled by the Regulator.

List of References

1. J. D. Salehi, Z-L. Zhang, J. F. Kuros, and D. Towsley, Supporting Stored Video: Reducing Rate Variability and End-to-End Resource Requirements through Optimal Smoothing. In *Proc. ACM SIGMETRICS*, pp. 14-26, May 1996.
2. J. Zhang and J. Hui. Applying traffic smoothing techniques for quality of service control in VBR video transmission. *Computer Communications*, special issue on building Quality of Service into Distributed Systems, 1997.
3. S. S. Lam, S. Chow, and D. K. Y. Yau. An Alorithm for Lossless Smoothing of MPEG Video. In *Proceedings, 1994 SIGCOMM Conference*, pp. 281-293, London, UK, August 31st – Septempber 2nd, 1994.
4. J. Zhang and J. Hui. Optimal smoothness results and approximation techniques for real-time VBR video traffic smoothing. *IEEE Real-Time System Symposium Proceedings*, pp. 253-263, 1997.
5. D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, pp. 368-379, April 1990.
6. A. Raha, S. Kamat, and W. Zhao. Guaranteeing end-to-end deadlines in ATM networks. *Proc. of IEEE ICDCS*, pp. 60-68, June 1995.
7. A. Raha, S. Kamat, and W. Zhao. Admisson Control for Hard Real-Time Connections in ATM LANs. In *Proceedings of the IEEE INFOCOM '96*, pp. 180-188, March 1996.
8. M. Grossglauser, S. Kashev, and D. Tse. RCBR: A Simple and Efficient Service for Multiple Time-Scale Traffic. *Proc. ACM SIGCOMM*, pp. 219-230, August 1995.
9. C. Partgidge. *Gigabit Networking*. Addison-Wesley, Reading, Massachusettes, 1994.
10. A. R. Reibman and A. W. Berger. On VBR Video Teleconferencing over ATM Networks. *Proc. IEEE GLOBECOMM*, pp. 314-319, 1992.
11. Vosaic Inc. Vosaic TVPlayer. http://www.vosaic.com.
12. Vivo Software. VivoActive Power Player. http://www.vivo.com.
13. Real Networks. RealPlayer audio player. http://www.real.com.
14. L. Delgrossi, C. Halstrick, D. Hehmann, R. G. Herrtwich, O. Krone, J. Sandvoss, and C. Vogt. Media scaling for audiovisual communications with the Heidelberg transport system. In *Proceedings of ACM Multimedia '93*, pp. 99-104, August 1993.

15. H. Kanakia, P. Mishra, and A. Reibman. An adaptive congestion control scheme for real-time packet video transport. In *Proceedings of ACM SIGCOMM '93*, pp. 20-31, September 1993.
16. P. Panacha and M. E. Zarki. Bandwidth requirements of variable bit rate MPEG sources in ATM networks. In *Proceedings of INFOCOMM '93*, pp.902-909, March 1993.
17. S. Cen, C. Pu, and J. Walpole. Flow and Congestion Control for Internet Media Streaming Applications. Technical Report CS-97-03, *Preceeding Multimedia Computing and Networking*, 1998 (MMCN98).
18. S. Cen, C. Pu, R. Staehli, C. Cowen, and J. Walpole. A Distributed Real-Time MPEG Video Audio Player. In *NOSSVAD'95, Lecture Notes in Computer Science 1018* (1995), 151-162.
19. Z. Chen, S.-M. Tan, R. H. Campbell, and Y. Li. Real time video and audio in the World Wide Web. In *Fourth International World Wide Web Consortium*, Boston, Massachusettes, December 1995.
20. R. H. Campbell, S.-M. Tan, Z. Chen, and D. Xie. Method of and System for Transmitting and/or Retreiving Real-Time Video and Audio Information over Performance-Limited Transmission Systems. Patent PCT/US96/19226.
21. W. R. Stevens. TCP/IP, Volume 1: The Protocols. Addison-Wesley, Reading, Massachusettes, 1994.
22. W. R. Stevens. TCP/IP, Volume 2: The Implementation. Addison-Wesley, Reading, Massachusettes, 1995.
23. J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. *MPEG Video Compression Standard*. Chapman & Hall, New York, 1997.
24. B. G. Haskell, A. Puri, and A. N. Netravali. *Digital Video: An Introduction to MPEG-2*. Chapman & Hall, New York, 1997.
25. R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications, and Applications*. Prentice Hall PTR, New Jersey, 1995.
26. P. Chan and R. Lee. *The Java Class Libraries*. Addison-Wesley, Reading, Massachusettes, 1997.
27. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusettes, 1995.
28. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusettes, 1997.
29. R. Martin, D. Riehle, and F. Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, Massachusettes, 1997.

30. M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Massachusettes, 1997.