PACKET LEVEL FRAME DISCARD FOR MPEG-2 VIDEO IN AN ACTIVE NETWORK

By

KYLE RUSSELL BROWN

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002
ACKNOWLEDGMENTS

I would like to extend thanks to my chairman, Dr. Randy Chow, and the other members of my supervisory committee, Dr. Jonathan Liu and Dr. Richard Newman, for their ideas and guidance on this project. Special thanks are extended to Rory J. De Simone, without whose support this thesis would never have been completed. Finally, I wish to thank my wife Pamela and daughter Erika for their continuing love and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
<tr>
<td><strong>CHAPTERS</strong></td>
<td></td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>MPEG-2 Video and Congestion</td>
<td>2</td>
</tr>
<tr>
<td>Goals</td>
<td>2</td>
</tr>
<tr>
<td>Novel Aspects</td>
<td>3</td>
</tr>
<tr>
<td>Overview</td>
<td>4</td>
</tr>
<tr>
<td>2 MPEG-2</td>
<td>5</td>
</tr>
<tr>
<td>Video Encoding</td>
<td>5</td>
</tr>
<tr>
<td>Frame Loss</td>
<td>6</td>
</tr>
<tr>
<td>Perceived Video Quality and Cost</td>
<td>8</td>
</tr>
<tr>
<td>3 ANTS ACTIVE NETWORK</td>
<td>10</td>
</tr>
<tr>
<td>ANTS</td>
<td>10</td>
</tr>
<tr>
<td>Capsules and Protocols</td>
<td>11</td>
</tr>
<tr>
<td>Node Operating System</td>
<td>12</td>
</tr>
<tr>
<td>Demand-Pull Code Distribution</td>
<td>14</td>
</tr>
<tr>
<td>4 IMPLEMENTATION OF AN ACTIVE ROUTER</td>
<td>16</td>
</tr>
<tr>
<td>Assumptions</td>
<td>17</td>
</tr>
<tr>
<td>“Regular” Method</td>
<td>21</td>
</tr>
<tr>
<td>“Active” Method</td>
<td>22</td>
</tr>
<tr>
<td>“Dispersed” Method</td>
<td>24</td>
</tr>
<tr>
<td>ANTS Classes</td>
<td>25</td>
</tr>
</tbody>
</table>


5 SIMULATION RESULTS........................................................................................................... 30
  “Regular” Method ............................................................................................................. 31
  “Active” Method ............................................................................................................. 32
  “Dispersed” Method ....................................................................................................... 34
  I-Frame Packet Density .................................................................................................. 37

6 CONCLUSION ..................................................................................................................... 39

BIBLIOGRAPHY ................................................................................................................... 41

APPENDICES

A FRAME LOSS .................................................................................................................... 43
B CAPSULE CLASS .............................................................................................................. 44
C PROTOCOL CLASS ......................................................................................................... 46
D APPLICATION CLASS .................................................................................................... 47

BIOGRAPHICAL SKETCH ................................................................................................. 67
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Capsule API</td>
<td>26</td>
</tr>
<tr>
<td>2. DataCapsule API</td>
<td>27</td>
</tr>
<tr>
<td>3. Protocol API</td>
<td>28</td>
</tr>
<tr>
<td>4. Application API</td>
<td>28</td>
</tr>
<tr>
<td>5. Regular method average costs</td>
<td>30</td>
</tr>
<tr>
<td>6. Active method costs</td>
<td>32</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Typical MPEG-2 frame dependencies</td>
<td>7</td>
</tr>
<tr>
<td>2. Entities in an ANTS active network</td>
<td>11</td>
</tr>
<tr>
<td>3. Janos architecture</td>
<td>13</td>
</tr>
<tr>
<td>4. Demand loading of capsule code</td>
<td>14</td>
</tr>
<tr>
<td>5. Packets and buffers</td>
<td>18</td>
</tr>
<tr>
<td>6. Congestion overlap in a two-router network</td>
<td>19</td>
</tr>
<tr>
<td>7. Regular method buffer size vs. cost</td>
<td>31</td>
</tr>
<tr>
<td>8. Active method size vs. cost</td>
<td>32</td>
</tr>
<tr>
<td>9. Percentage cost decrease—regular to active method</td>
<td>33</td>
</tr>
<tr>
<td>10. Regular vs. active methods</td>
<td>34</td>
</tr>
<tr>
<td>11. (12, 3) Active vs. dispersed, buffer = 100</td>
<td>35</td>
</tr>
<tr>
<td>12. (12, 3) Active vs. dispersed, buffer = 400</td>
<td>35</td>
</tr>
<tr>
<td>13. (18, 6) Active vs. dispersed, buffer = 100</td>
<td>36</td>
</tr>
<tr>
<td>14. (18, 6) Active vs. dispersed, buffer = 400</td>
<td>36</td>
</tr>
<tr>
<td>15. (18, 6) I-frame size comparison, active</td>
<td>37</td>
</tr>
<tr>
<td>16. (18, 6) I-frame size comparison, dispersed</td>
<td>38</td>
</tr>
</tbody>
</table>
Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

PACKET LEVEL FRAME DISCARD FOR MPEG-2 VIDEO IN AN ACTIVE
NETWORK

By

Kyle Russell Brown

May 2002

Chairman:  Yuan-Chieh Chow
Major Department:  Computer and Information Science and Engineering

Active networks are a new paradigm for networking.  Allowing customized
computation within the network provides users and applications with the ability to
provide services not possible in a traditional network.  MPEG-2 video can benefit from
this new style of network by altering the way routers within a network react when
congestion occurs.  We can greatly improve the perceived quality of delivered MPEG-2
video by selectively discarding frames during times of congestion.  Doing so within the
network eliminates the latency of feedback controlled server side adaptations.  Reaction
to congestion is nearly instantaneous and restoring after congestion happens just as
quickly.
CHAPTER 1
INTRODUCTION

The MPEG-2 standard for video compression has become one of the predominant forms of video encoding used to transport video in today’s Internet. Although still in its infancy, video over the Internet is becoming increasingly popular. Increased market penetration of broadband technologies such as ADSL and cable modems has allowed an ever-increasing number of consumers to use streaming digital video applications. These applications range from distance learning to video teleconferencing to simply watching today’s news or sports headlines on demand. As the number of applications and the number of broadband consumers grow, the bandwidth demand on the Internet will grow as well. The challenge is to support this growth while keeping the perceived quality of the delivered video high. This can be done using compression algorithms such as MPEG-2, yet there are times when congestion will occur and video frames will be lost. One way to minimize the cost of these losses is to selectively discard frames based on their contribution to picture quality. This can be done at the video server but it is hard for the server to anticipate network congestion. Reacting to congestion may take too long when delivering streaming video, especially when the video is an interactive video conference [Tal94]. An active network can solve this problem by moving the frame discard algorithm into the network to the exact places where congestion occurs, and dropping the frames in a per-packet, application-aware way. In this way, we provide the
highest quality video possible while at the same time freeing up scarce network resources.

**MPEG-2 Video and Congestion**

Streaming digital video, especially if it is live, is extremely sensitive to network congestion. MPEG-2 video frames are divided into packets for delivery across the Internet with each individual packet finding its way to the desired destination. In this environment, packets will be dropped. The way in which we react to these losses will determine the final quality of the delivered video. Exacerbating this problem is the fact that MPEG-2 video is commonly transmitted using the User Datagram Protocol (UDP) instead of the more reliable Transmission Control Protocol (TCP) because of the time sensitive nature of streaming video. The process of providing reliable transmission via TCP simply takes too long for it to be useful in delivering streaming video. UDP’s “best effort” policy places less stress on network resources by eliminating acknowledgements and retransmissions. Since a dropped packet is of no use if it is retransmitted and subsequently delivered too late, UDP is a good match for MPEG-2 video delivery.

**Goals**

The goal of this thesis is to show that employing an algorithm that selectively discards MPEG-2 packets can increase perceived quality of MPEG-2 video. We will show that by using active network technology we can deploy the algorithm at the exact place in the network where the congestion occurs. We will show quantitatively that the perceived quality of the resulting video is higher than without these improvements.
Novel Aspects

Other work that used an active network to implement frame level dropping in MPEG-2 video was the technical report by Bhattacharjee et al. [Bha98]. This implementation is different in several important ways. First, our algorithm is implemented using ANTS. ANTS is a widely used tool used for the research and development of active networks. Second, a frame-dropping algorithm is used that is aware of the dependencies between the different types of MPEG-2 frames. This algorithm is not unique, however. Many different algorithms were explored by Zhang et al. [Zha99], and the one used here is a slightly modified version of their selective frame discard heuristic. Third, Zhang et al. [Zha99] also used a cost function, which has been modified slightly for use here. A great deal of research has been done attempting to quantify perceived Quality of Service (QoS) in packet video delivery. The cost function uses the fact that a sequence of discarded frames is more costly than the loss of an equal number of single frames spread out over a length of time. Bhattacharjee et al. [Bha98] did not quantify their cost in terms of perceived quality. Fourth, I use an algorithm that takes into account the fact that MPEG-2 frames are different sizes and thus must be implemented at the packet level to attain the greatest efficiency. The work done by Zhang et al. [Zha99] is the most closely related to this work in terms of MPEG-2 frame discard. However, they did not consider frames at the packet level and implemented all their algorithms at the server. They did not consider an active network implementation of their work. Bhattacharjee et al. [Bha96] implemented a frame dropping algorithm in an active network, but they considered solely the effect of reducing bandwidth consumption as interacting MPEG streams had their frames dropped.
Overview

The next chapter describes MPEG-2 video encoding. Chapter 3 introduces active networks and describes the ANTS implementation used to simulate an active network in this thesis. Chapter 4 describes the proposal for improving streaming MPEG-2 video and the implementation using ANTS. Chapter 5 describes the simulations performed and discusses the results. Chapter 6 concludes.
CHAPTER 2
MPEG-2

This chapter gives a brief background on MPEG-2 video encoding

**Video Encoding**

The MPEG-2 video stream is comprised of three types of images: I-, P- and B-frames [Ste95]. I-frames (intra-coded) are coded without reference to any other image. I-frames are independent and require no other frame for decoding. Thus, they are the points of access when randomly accessing a position within the video stream. I-frames have the least amount of compression and so are the biggest in terms of file size. P-frames (predictive-coded) reference either the previous I-frame and/or all the previous P-frames for coding and decoding. P-frame encoding is based on temporal redundancy. From frame to frame, an image usually does not completely change; it shifts, so individual blocks can be predicted based on previous blocks and motion estimates. P-frames hold the middle ground as far as compression. They generally are less than half the size of an I-frame, but still 3 times the size of a B-frame. B-frames (bi-directionally predictive-coded) reference the previous and following I- and/or P-frame. Bi-directional prediction allows B-frames to attain the highest compression ratio.

The MPEG video stream is comprised of six layers. The highest level is the sequence layer, followed by the group of pictures layer, the picture layer, the slice layer, the macro block layer, and the block layer. When MPEG-2 video is sent over a network, it is the group of pictures (GOP) layer that is most significant. A GOP consists of one I-
frame followed by a combination of P- and B-frames. A typical GOP might contain the following sequence of frames: I B B P B B P B B P B B. The order and combination of the P- and B-frames determine the overall compression rate and conversely the quality of the resultant video. For any one video, higher compression will result in lower perceived quality. Variable rate encoding is allowed so that perceived quality can be maintained at a constant level while the amount of motion in a video varies. This variable rate encoding is accomplished by altering the ratio of I-, P-, and B-frames to change the overall compression rate. For example, parts of a news broadcast with a lot of motion, such as highlights from a basketball game, can be encoded with a smaller GOP. This provides for more total I-frames and fewer P- and B-frames for less overall compression. Likewise, when an anchorperson is talking and almost motionless, the GOP can be much bigger. There will be a greater distance between I-frames, a few P-frames and a greater number of B-frames. This results in far greater compression with approximately the same perceived quality. The notation used to describe a GOP uses the two variables (N, M). N is the distance in frames between successive I-frames. M is the distance in frames between successive P-frames. The typical GOP described above has \( N = 12 \) and \( M = 3 \). The number of frames per GOP is N. The number of P-frames per GOP is \( (N / M) - 1 \). The number of B-frames between I- or P-frames is M-1. For example a (18, 6) GOP would have the following sequence: I B B B B P B B B B B P B B B B B.

**Frame Loss**

When individual frames are lost in an MPEG video stream, the amount of information lost is dependent on the type of frame that is lost. A lost B-frame will result in the smallest loss since it is the most highly compressed and has the smallest file size.
Also, no other frame is dependent on a B-frame, so when a B-frame is lost it does not affect any other frames. This is not true of I- and P-frames. When a P-frame is lost, 3 times the information is lost just based on the relative file size of a P-frame. In addition, because of the dependencies between frames, the surrounding B-frames will be lost as well. Any P-frame that follows the lost P-frame up until the next I-frame will also be useless. This type of error is known as temporal propagation [Ver98a]. In general, the loss of a P-frame results in the loss of the dependent B-frames preceding it and all the frames after it until the next I-frame. The loss of an I-frame is even more devastating. Because of direct and transitive dependencies, the entire GOP is lost when the I-frame is lost. Because of the size of I- and P-frames, they must be broken into pieces for transmission over the Internet. Because of this, the chance that a frame is lost is
increased because the loss of any one packet of a frame results in the loss of the entire frame.

Figure 2-1 shows a (9, 3) GOP with the following sequence: I₁ B₁ B₂ P₁ B₃ B₄ P₂ B₅ B₆. Frame B₂ is dependent on I₁ and P₁. Frame P₁ is dependent on just I₁. Frame P₂ is dependent on both I₁ and P₁. Because of the transitive nature of the dependencies, one can see that frame B₅ would be lost if P₁ was lost since B₅ depends on P₂ which depends on P₁.

Perceived Video Quality and Cost

A common way to describe the quality of video is peak signal-to-noise ratio (PSNR). However, the way in which humans perceive visual information is based on the concepts of contrast sensitivity and masking [Lam96]. PSNR does not consider these concepts. Even though each pixel in error causes a reduction in PSNR, the human visual system, being much more complex and dependent on viewing conditions, may not register any noticeable difference. The perceptual quality saturates at high bit rates and this is not captured by the PSNR measure [Ver98a]. Therefore, other metrics that are based on the human visual system have been developed. These include the moving pictures quality metric (MPQM) [Lam96], the Sarnoff JND (just noticeable difference) vision model [Lub97], and the perceptual distortion metric (PDM) for digital color video [Win98]. These metrics have been shown to behave consistently with human judgment [Ver98b]. However, we are using a frame-based discarding algorithm, and the above methods do not transfer well to our case.

To quantify the video quality in a frame-based system we must derive a cost based on the number of frames that are discarded. The cost function we use is that
described by Zhang et al. [Zha99]. Two costs are described. The first is the length of a sequence of consecutive discarded frames. The second gives the cost of a single discarded frame based upon its distance from the most recently discarded frame. Since we are describing frame loss that occurs in bursts, the first metric is the one that will apply most often. If frame \( i \) belongs to a sequence of discarded frames, the cost \( c_i \) is defined to be \( l_i \), if frame \( i \) is the \( l_i^{th} \) consecutively discarded frame in the sequence. The total cost of \( i \) consecutive frames is then \( c = \frac{i^2 + i}{2} \). If frame \( i \) is a single discarded frame and \( d_i \) is the distance to the previous discarded frame, the cost is then \( c_i = 1 + \left( \frac{1}{\sqrt{d_i}} \right) \). The total cost of congestion is the total of the costs of each sequence of consecutive frame losses plus the costs of any individual frame losses.
CHAPTER 3
ANTS ACTIVE NETWORK

Today’s typical network has a passive store and forward architecture. The routers inside the network simply look at the packet header to extract the destination address, look up the next hop in a table, and send the packet on its way. In an active network, the routers can be programmed on the fly by the active capsules moving through them. This provides many benefits including allowing a range of new applications that can leverage computation within the network, and increasing innovation by decoupling the services provided by a network from its underlying infrastructure [Ten96].

ANTS

An active network consists of many entities working in concert. ANTS (Active Node Transfer System) is composed as shown in Figure 3-1. Special types of packets called capsules are sent and received by applications through programmable routers called active nodes. The applications can receive special network services by requesting them using the capsules. Active nodes are linked together with link layer channels. Active nodes are compatible with existing routers to allow incremental deployment of an active network in the Internet. There are three key components in the ANTS architecture: capsules and protocols, active node operating system, and demand-pull code distribution.
Capsules and Protocols

A capsule is analogous to a packet in an IP network. The format of a capsule must be compatible with both active and IP networks. The header is configured so that IP routers see the packets as IP packets and forward them accordingly. This way IP routers appear as active routers with no services available. The most important information the active network portion of a capsule header contains is the type, which identifies the code group, protocol, and forwarding routine associated with that capsule. The type is kept compact and secure by producing it using an MD5 message digest algorithm. A forwarding routine is used at each node to process each capsule. Some forwarding routines are “well-known” and are available at every active node. Other routines are specific to a particular application and rely on the code distribution mechanism to make them available to a capsule. A code group is a collection of related capsule types. The code distribution mechanism transfers each groups forwarding
routines as a unit. A protocol is a collection of related code groups that are treated as a single unit by the node. Capsules within the same protocol will typically share information within the network [Wet98]. In order to define a new service, one must define a new protocol by adding a capsule to a new code group and a new protocol.

**Node Operating System**

A new network service must acquire resources at each active node it encounters. A generic operating system (OS) such as Unix would not provide the node with the proper protection against runaway protocols and would not provide independent code groups the necessary separation from other groups while still providing each group an independent view of the network. A custom OS can provide custom sharing and protection mechanisms in a way that is compatible with high levels of performance. The latest version of ANTS uses Janos, which is a java-oriented OS for active network nodes that was developed at the University of Utah [Tul01]. The software at an active node is logically divided into three layers: the node OS, the execution environment (EE), and the active application (AA) layers. The node OS provides the node with low-level resource management and mediates the demand for transmission, computing, and storage. The node OS has local significance only, managing the resource of just one node. The EE sits on top of the node OS and provides a programmable network application programming interface (API). Typically, the network API defines a virtual machine, which is available to the network programmer. A node can support multiple EE’s simultaneously. This greatly reduces the amount of standardization needed to implement an active network and therefore speeds up the implementation process. The top layer contains the AA’s. An AA accomplishes some desired communication function using a combination of packet
forwarding and computation in the network nodes. The architecture of the Janos system is shown in Figure 3-2. The ANTS runtime runs atop the JanosVM, which is a modified, resource conscious Java virtual machine. These two components constitute the EE and run on Moab, the University of Utah’s node OS. Moab is built on their OS component library, the OSKit.

![Janos Architecture Diagram](image)

**Figure 3-2 Janos architecture [Tul01]**

Janos has four main goals. The first is untrusted code support. An AA must only be allowed to see packets that it is allowed to see and send packets that it is allowed to send. AA’s must terminate quickly or be preemptible. AA’s must not be allowed to interfere with other user code (other AA’s) or the OS itself. Janos must be able to terminate any user code. The second goal is resource management. AA’s must be limited on the amount of memory they consume and Janos must reclaim unused memory. CPU time must be regulated so that no AA can dominate CPU usage. Outgoing network bandwidth must be constrained so that finite resources are shared and no one user has
dominance over another. The third goal is performance. Janos tries to demonstrate that flexible active nodes can be fast and efficient for appropriate uses. The fourth goal is separable components. Although all the Janos components are designed to work together, each should be useful independently [Tul01].

**Demand-Pull Code Distribution**

Two schemes are apparent when deciding how to transfer code from node to node. At one extreme, the code could be transferred along with every capsule in the capsule’s payload. This would only work with very small programs and would result in a lot of wasted bandwidth in situations where a stream of packets all used the same code. At the other extreme, code could be preloaded at the nodes using an out-of-band approach. This approach does not allow for rapid and decentralized deployment and would require an enormous amount of space at the nodes as more and more services are added over time. ANTS uses a hybrid of the two approaches. The distribution of code is an in-band function that distributes code as needed, but not within the actual packet. This

![Diagram](image-url)
limits the distribution of code to precisely where it is needed. The code is kept in a cache. If a stream of capsules of the same type is processed at a node, the required code will be in the cache. The demand-pull protocol is illustrated in Figure 3-3.

In step one, a capsule is forwarded. In step two, the capsule arrives at the next node, where the required code is not present, which causes a load request to be sent to the previous node. In step three, the load request arrives at the previous node and a series of load responses are sent to the requesting node. Finally, in step four, the load responses arrive at the requesting node and are assembled to obtain the required code. The protocol works because the previous node will always have the code necessary. Without it, the previous node would not have been able to forward the capsule in the first place. As code is demand loaded along a path, the capsules of that type no longer need to have code loaded for them, it is already present at all the nodes along the path once the first capsule of that type has been delivered. If the network path changes, the demand loading process will start again for the new path.

---

1 Because the cache is limited in size, two different capsule types being sent over the same network at the same time could cause thrashing if the code size of each is large enough to displace the other capsule’s code.
CHAPTER 4
IMPLEMENTATION OF AN ACTIVE ROUTER

In each active node, we have a buffer that holds packets during network congestion. During this congestion, the buffer fills up. When the congestion is over, a burst of traffic from that node will result as it empties its buffer. During normal operation, packets do not reside in the buffer. We use three methods for comparison. The first method, the “regular” method, is based on a passive network router. The passive network approach to discarding MPEG-2 video frames is equivalent to having no approach at all. Congestion causes the buffer to fill up and any frames entering the node after that are lost indiscriminately. This method is used as a baseline for the performance of the second method, the “active” method. An active network approach allows us to discriminate between types of frames when deciding what to discard. When congestion occurs at an active node the application is able to selectively discard frames. Because we operate at the packet (capsule) level, we can fine-tune the discard of frames to a very fine level. The third approach, the “dispersed” method, uses the same algorithm as the active approach, but adds the ability to discard B-frames from the buffer in an intelligent manner. We must first make some assumptions to facilitate a constrained test.
Assumptions

Loss Bursts

In today’s networks, the propagation channel (copper wire, coaxial cable, or optical) is not likely to cause a transmission loss. Losses and delays are usually at the routers and switches within the network. Outgoing bandwidth constraints and processing limitations cause buffers to fill up and overflow. This causes most packet losses to occur in bursts [Ver98a]. For that reason, we do not consider the case in which individual packets in an MPEG-2 stream are dropped. It is easy to see that if a single packet is dropped from an I-frame, the whole GOP should be discarded to conserve bandwidth. Similar rules can be adopted for P and B-frames [Ram93]. We could use automatic retransmission request (ARQ) to obtain the lost packet, but the delay involved is too great. The retransmitted packet would arrive too late to be decoded. There is also a multitude of other ways to correct or conceal missing video information. These include forward error correction (FEC), forward error concealment, error concealment by postprocessing, and interactive error concealment [Wan98]. Our thesis is that when confronted with congestion and buffer overflows at a router, we can greatly improve the perceived quality of the video by selectively choosing which packets to discard. We are discarding packets only when absolutely necessary, as opposed to estimating network conditions or waiting for feedback from the network and then making changes at the video server.

Buffer Size

In an active node, there is a tradeoff between a smaller buffer and a larger one. A small buffer does not allow enough room to handle periods of significant congestion.
Since video by nature has time constraints, we also do not want the buffer to be so large that it would adversely affect the timely arrival of packets to their destination. Imagine a network as in Figure 4-1.

![Figure 4-1 Packets and buffers](image)

A packet stream enters a network from a sending node. A single buffer in the network experiences congestion and accumulates a large number of MPEG-2 capsules. A burst of capsules results when the congestion is over. Assuming the buffer of the MPEG-2 playback application at the receiver was full at the start of the congestion, it should have been reduced by the same amount that has accumulated in the network buffer. The receiver will have no disruption if both 1) the network buffer did not discard any capsules and 2) the buffer in the receiver is at least as large as the network buffer. If it is smaller than the network buffer, the burst of packets from a full network buffer will overwhelm the receiver and frame loss will result.

**Multiple Nodes**

Multiple buffers within the network can be viewed as an extension of this basic example. Any time a burst of packets is received at a node that is experiencing congestion, the situation mirrors that of an MPEG-2 receiver. It is conceivable that more than one node within a network would become uncongested at the same time. With no other delays in the network, these two bursts would arrive at the receiver almost
simultaneously. Thus, some or all of the total delay within the network could combine to cause a buffer in the receiver to overflow. Obviously, with only one active node, we can have a much larger buffer than we can have in a network with many active network nodes. Optimal buffer size is therefore an open question, and various buffer sizes are explored in the implementation. The size of the buffer is in terms of the number of capsules that it can hold.

In a real network, there would of course be more than one intermediate node. To see the effect of this, we can imagine a network with four nodes (sender, router 1, router 2, and receiver). The routers are independent of each other and so the amount of delay within the network depends heavily on the timing of the individual nodes’ delays. When the delays coincide with each other, the cost of the delay is the cost in just one node. No

![Figure 4-2 Congestion overlap in a two-router network](image)

Figure 4-2 Congestion overlap in a two-router network
packets are being received in the second node, so no packets are lost. As the overlap decreases, the cost goes up. Figure 4-2 illustrates the situation when the overlap of two routers is 50%.

The first router buffers as many packets as it can then discards packets until the congestion is gone. It then forwards the buffered packets to the second router. Due to congestion, the second router is unable to forward those packets and so buffers them. This fills up the buffer in the second router. Packets are being forwarded by the first router and must be discarded by the second router until the congestion there is gone. When the congestion is gone at the second router, it sends the packets in its buffer and then continues normally. The only packets saved are those originally buffered by the first router. All packets from that point on are lost until the second router’s congestion is over. The effect is that the congestion is cumulative. The receiver simply sees a long delay with no packets received followed by a burst of buffered packets and then normal delivery. When the delays no longer overlap, their costs are independent and the total cost for the network is the sum of the costs at the individual nodes. Thus, the effects that are seen in our example with only one intermediate node can be extrapolated to situations where multiple nodes are present.

**Group of Pictures**

We use a consistently sized GOP that is 12 frames long for the majority of the simulations. The order of the frames is shown below. The first subscript given for each frame is the GOP number. The second subscript is the sequence number of that frame type within each GOP. For example, a (12, 3) GOP would look like the following:
For simplicity of exposition, frames are transmitted in decode order. The decode order (and thus the transmission order) is:

\[ I_1 B_{11} B_{12} P_{11} B_{13} B_{14} P_{12} B_{16} P_{13} B_{17} I_2 B_{21} B_{22} P_{21} B_{23} B_{24} P_{22} B_{25} B_{26} P_{23} B_{27} B_{28} I_3 \ldots \]

Note that the decode order of the first frame is different from all the subsequent frames because of the dependence of \( B_{17} \) and \( B_{18} \) on \( I_2 \).

**Packets per Frame**

In the paper by Boyce and Gaglianello [Boy98], Tables 1 and 2 give statistics on 1Mbps and 384Kbps MPEG-2 video. Using this data, we can calculate an approximate number of packets per frame assuming that each packet contains as much information as a B-frame packet. This ratio of I-frame size to P-frame size to B-frame size is 8:3:1. If a B frame fills one packet, then an I-frame fills eight packets and a P-frame fills three packets. We assume a frame rate of 27 frames per second. With that frame rate and the above GOP layout, we have 25 packets per 12 frames. Thus, we have 56.25 packets per second. Therefore, a 1000ms buffer in our active node will contain 56 packets.

**“Regular” Method**

The regular method is similar to what would happen in a non-active node. Any packet that enters the node after the buffer is full will be discarded and that information will be lost. Assuming we have a 1000ms buffer, 1 second of congestion will result in no packet loss because the buffer is just big enough to handle the backup. However, 2 seconds of congestion will result in the loss of 56 consecutive packets. The amount of frame loss is dependent on the position in the packet stream of the first packet lost and
the number of packets lost. In the best case of a 56 packet loss, we would lose the last 6 packets (equaling 3 B-frames and one P-frame) plus the next two consecutive GOPs (24 frames) plus two B frames that would be received but could not be decoded because they are dependent on frames lost in the previous GOP. 30 frames would be lost. In the worst case, we would lose three GOPs plus quite a few dependent frames for a total loss of 41 frames. Either way, a two second loss of the network would result in more than the one second’s worth of frame loss that you would naively expect (27 frames).

To assign a cost to this loss of frames we use the formulas for cost given in chapter 2. Since the point in the packet stream at which loss begins is random, we calculate an average frame loss by calculating the frames lost using each of the 25 possible starting points within the (12, 3) GOP. The 56-packet loss in the example above has an average loss of 36.32 frames. To calculate losses when the congestion consumes any number of lost packets, we can make a table like that shown in Appendix A. From this table we derive the linear function $frames\ lost = (packets\ lost * .48) + 9.44$ which will give the number of frames lost for any number of packets lost (greater than 25).

“Active” Method

Using the “active” method, the first second of congestion is the same as the regular method, with the buffer filling up. After that, the difference is substantial. The idea is that B-frames are the least important, they carry the smallest amount of information and have the greatest dependence on the other frames, for that reason they are the first ones to be dropped. They are dropped in a first-in-first-out (FIFO) order. Similarly, P-frames are dropped next, with the individual frames dropped in order of highest subscript first. The I-frames are dropped last. They are the only frames that
independently contain enough information to create an entire image. This method will drop more frames than the regular method, but more usable information will be retained, and we will show that the frames that are left can be combined to produce a higher quality motion picture than we would have if frames were dropped indiscriminately.

The algorithm is as follows.

If the queue is not full, enqueue the incoming packet.
else {
    Check the type of the incoming packet.
    If the incoming packet is a B-frame, drop it.
    Else
        If the incoming packet is a P-frame {
            If there is a B-frame packet in the queue, drop it and enqueue the incoming P-frame packet
            Else
                If there is a P-frame packet in the queue, check the subscript
                    Drop the P-frame packet with the highest subscript, either from the queue or the incoming P-frame packet (if they are equal drop the incoming packet)
                Else
                    Drop the incoming P-frame (there are only I-frames in the queue)
        }
        Else
            If the incoming packet is an I-frame {
                If there is a B-frame in the queue, drop it and enqueue the incoming I-frame
                Else
                    If there is a P-frame in the queue, drop the one with the highest subscript and enqueue the incoming I-frame
                Else
                    Drop the incoming I-frame (once the queue has only I-frames, we drop all incoming packets)
            }
    }
}
Using the algorithm above with the simple B-frame discard, a typical 2-second congestion would produce the following. We assume the congestion starts at the leading edge of a GOP. The first 56 packets are enqueued and then the individual packets are discarded or enqueued according to the above algorithm. At the end of the 2 seconds, the queue will contain the following frames: \( I_1 P_{11} I_2 P_{21} I_3 P_{31} I_4 P_{41} I_5 P_{51} \).

The last frame dropped is \( B_{48} \). 33 B-frames were dropped, 8 P-frames were dropped and 0 I-frames were dropped. The total number of frames dropped is 41. Notice that this is the same number of frames lost in the worst case of the regular algorithm. However, what is important is the perceived quality of the video that remains and for this we use the cost function.

Applying the cost function to the regular and active methods above, we see that the regular method has an average frame loss of 36.32 frames. These frames are consecutive. Therefore, the total cost is 678. The active method loses 41 frames, but not in a row. The largest loss is between the P-frames and the I-frames that follow. Here \( i = 8 \) and the cost is 36. This happens five times. The other loss is between the I-frames and the P-frames that follow. Here \( i = 2 \) and the cost is 3. This also happens five times. Thus, the total cost of the active method is 159. This is a 76.5% improvement over the regular method.

**“Dispersed” Method**

Since groups of lost frames typically cost more than individual lost frames, one possible improvement to the active method might be to distribute the lost B-frames as much as possible. The active method uses a simple FIFO strategy when deciding which B-frames to discard from the buffer. This does not take advantage of the cost function.
which says the greater the distance between individual lost frames the lower the cost.

Since the B-frames are the first ones discarded from the queue, it makes sense to try to improve this area. We disperse the B-frames that are removed as much as possible. In order to implement this we label all the B-frames in the queue according to their position, and discard them as needed based on their label. We strive to maximize the distance of the frame being discarded from previously discarded frames.

As an example, we look at a (12, 3) GOP using a 56 packet buffer. For small numbers of lost B-frames the advantage of the dispersed method is significant. If two B-frames are lost and they are in consecutive GOPs, they could have up to 21 frames distance. The cost of losing these two frames would then be 2.22.\(^1\) The cost of losing two adjacent frames is 3. This is a considerable (26\%) improvement. However, as the number of B frames lost increases, this improvement quickly deteriorates. A loss of seven B-frames from the same two consecutive GOPs would result in the simpler FIFO algorithm for B-frames having an 8.5\% advantage over the more complicated dispersed algorithm. The average over all the cases shows the FIFO algorithm having a 3.6\% advantage over the more complicated dispersed algorithm. We explore the effects of buffer size, GOP type, and amount of congestion on the relative advantage/disadvantage of the dispersed method in the simulations.

**ANTS Classes**

The ANTS toolkit maps active nodes, channels, capsules, extensions, and applications to Java classes. Active nodes are represented by an instance of the class `Node` and may act as a router or end-system that just sends and/or receives capsules. A

\(^1\) C = 1 + (1 + 1/sqrt(21)) = 2.22. Please refer to the cost metric discussed in Chapter 2
Node exports its services to capsules to evaluate their forwarding routines, applications to register protocols and send and receive capsules, and node extensions to augment local node services. These exported services are logically separate so that the API available to capsules is not available to applications and vice versa. To write a new network service for ANTS, one must write subclasses of the Capsule, Protocol, and Application classes [Wet97].

**Capsule Class**

There are two classes available when writing a subclass of Capsule, the minimal class Capsule and the DataCapsule class, which provides for UDP-like service between applications. Certain methods must be overridden when writing a capsule class. The most important of these is the `evaluate()` method. It is invoked at every node the capsule visits and can receive the services available to capsules at that node. The capsule API and the DataCapsule API are given in Table 4-1 and Table 4-2.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getSrc()</code></td>
<td>Get source address</td>
</tr>
<tr>
<td><code>int getDst()</code></td>
<td>Get destination address</td>
</tr>
<tr>
<td><code>void setDst(int address)</code></td>
<td>Set destination address</td>
</tr>
<tr>
<td><code>int getResources()</code></td>
<td>Get remaining resources</td>
</tr>
<tr>
<td><code>void prime(Capsule parent)</code></td>
<td>Transfer resources to fresh capsule</td>
</tr>
<tr>
<td><code>int getPrevious()</code></td>
<td>Get previous active node address</td>
</tr>
<tr>
<td><code>byte[] getCapsuleID()</code></td>
<td>Get capsule type</td>
</tr>
<tr>
<td><code>byte[] getGroupID()</code></td>
<td>Get code group type</td>
</tr>
<tr>
<td><code>byte[] getProtocolID()</code></td>
<td>Get protocol type</td>
</tr>
</tbody>
</table>
The code for my `Capsule` class is given in Appendix B. This capsule class subclasses the `DataCapsule` class. Additional instance variables were added for the frame type, the per frame sequence number, the intra-frame sequence number, and the capsule’s destination. The `encode()`, `decode()`, and `length()` methods convert them via the `Xdr` class into an external representation that is suitable for transmission across the network. The `evaluate()` method simply delivers the capsule to all nodes except the sending node. It is up to the application running on the node to determine when and if to forward the capsule towards its destination.

**Protocol Class**

The `Protocol` class is straightforward since there is only one type of capsule class. This class is wrapped by a single code group that forms the entire protocol. The `Protocol` API is given in Table 4-3. The code for the `Protocol` class is given in Appendix C.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Short getSrcPort()</code></td>
<td>Get source port</td>
</tr>
<tr>
<td><code>void setSrcPort(short port)</code></td>
<td>Set source port</td>
</tr>
<tr>
<td><code>Short getDstPort()</code></td>
<td>Get destination port</td>
</tr>
<tr>
<td><code>void setDstPort(short port)</code></td>
<td>Set destination port</td>
</tr>
<tr>
<td><code>ByteArray getData()</code></td>
<td>Get payload</td>
</tr>
<tr>
<td><code>void setData(ByteArray data)</code></td>
<td>Set payload</td>
</tr>
<tr>
<td><code>DataCapsule(short sp, short dp, int da, ByteArray p)</code></td>
<td>Constructor</td>
</tr>
</tbody>
</table>
Table 4-3 Protocol API [Wet99a]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void startProtocolDefn()</td>
<td>Begin new protocol</td>
</tr>
<tr>
<td>void endProtocolDefn()</td>
<td>End current protocol</td>
</tr>
<tr>
<td>void startGroupDefn()</td>
<td>Begin new code group</td>
</tr>
<tr>
<td>void endGroupDefn()</td>
<td>End current code group</td>
</tr>
<tr>
<td>void addCapsule(String name)</td>
<td>Add capsule to current group</td>
</tr>
<tr>
<td>void addHelperClass(String name)</td>
<td>Add non-capsule to current group</td>
</tr>
</tbody>
</table>

Table 4-4 Application API [Wet99a]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void attachNode(Node n)</td>
<td>Connect to local node</td>
</tr>
<tr>
<td>Node getNode()</td>
<td>Get connected node</td>
</tr>
<tr>
<td>Short getPort()</td>
<td>Get connected port</td>
</tr>
<tr>
<td>int getDefaultResources()</td>
<td>Get resource limit</td>
</tr>
<tr>
<td>void setDefaultResources(int l)</td>
<td>Set resource limit</td>
</tr>
<tr>
<td>void register(Protocol p)</td>
<td>Register protocol</td>
</tr>
<tr>
<td>void unregister(Protocol p)</td>
<td>Unregister protocol</td>
</tr>
<tr>
<td>void send(Capsule c)</td>
<td>Send capsule via node with default resources</td>
</tr>
<tr>
<td>void send(Capsule c, int l)</td>
<td>Send capsule via node</td>
</tr>
<tr>
<td>void receive(Capsule c)</td>
<td>Receive capsule from node</td>
</tr>
</tbody>
</table>

Application Class

The Application class is the independent entity that is running on the individual nodes. Applications must be able to send and receive capsules, register/unregister protocols, and connect to the local node. The Application API to do these things is given in Table 4-4.

The start() method registers the MpegProtocol with the local node. At this point, we check to see if this node is to be a router, the final destination, or is the sending node. If it is the sending node, we simply start a thread that runs the MpegApplication run() method on this node. It is this run method that sends a warm-up capsule to get the protocol running on all nodes. Then capsules are created with
appropriate information to identify them as MPEG capsules and they are injected into the network in decode order as described in earlier. After each capsule is sent, the node is put to sleep for a short time to simulate the frame rate of MPEG video.

If the node is to be a router or receiver an instance of the appropriate class is instantiated on the node. A capsule will be directed to the receive() method in either case. In this method, a node designated as a receiver will simply perform some performance calculations and shut down when almost all of the capsules have been received. A node designated as a router will do one of three things.

A regular router will send a capsule toward its destination if there is no congestion. If there is congestion, the router will buffer the capsule if the buffer is not full. If the buffer is full, the capsule will be dropped. This functionality is designed to imitate a router that is not active. The size of the buffer and the length of the congestion period (in capsules) are defined in the configuration file and can be altered as needed. The congestion is started at a random capsule between capsule number 25 and number 49 inclusive. An active router will send the capsule toward its destination if there is no congestion and buffer it if the buffer is not full. When the buffer is full, however, the router will implement the algorithm outlined in earlier in this chapter. Once the congestion is over, the router will immediately send all the packets in its buffer, in order, with no delay between packets. An active router that is implementing the dispersed method behaves the same as the active method except that when a B-frame is to be discarded from the buffer, one is chosen that as distant as possible from the closest previously discarded B-frame. The code for the three Application classes is given in Appendix D.
CHAPTER 5
SIMULATION RESULTS

Three sets of simulations were done corresponding to the three Application classes: regular, active, and dispersed. Since the final cost is dependent on the place within a GOP at which the delay starts, the starting point for the delay is determined by a pseudo-random function. Each simulation was then run ten times. The results shown are the average of 10 runs. The difference between the regular and active methods is explored by varying the buffer size and the amount of congestion. The dispersed method is compared to the active method by varying the type of GOP, the buffer size, and the amount of congestion. A (12, 3) GOP, which has a relatively small number of B-frames, was compared that a (18, 6) GOP, which has a much larger number of B-frames. Another set of simulations was done to see the effects of I-frame size. We have previously assumed that an I-frame consists of 8 packets. We explore the effect of doubling the size of the I-frame to 16 packets on the active and dispersed methods.

<table>
<thead>
<tr>
<th>COST</th>
<th>buffer size</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 capsule delay</td>
<td>1040 +/- 235</td>
<td>587 +/- 154</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 capsule delay</td>
<td>4242 +/- 501</td>
<td>3336 +/- 405</td>
<td>1696 +/- 257</td>
<td>576 +/- 165</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300 capsule delay</td>
<td>9850 +/- 670</td>
<td>8403 +/- 528</td>
<td>5553 +/- 552</td>
<td>3485 +/- 256</td>
<td>1619 +/- 188</td>
<td>588 +/- 153</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
“Regular” Method

The regular method was simulated with three nodes (sender, router, and receiver) and congestion periods (delays) of 100, 200, and 300 capsules. The resultant video stream at the receiver was analyzed and the cost of the video degradation calculated. When buffer size is greater than or equal to the amount of congestion the cost is zero since there is no loss of video frames. The values given in Table 5-1 have a confidence interval as indicated. Figure 5-1 is a chart of those figures and shows that a larger buffer will always lower the cost of congestion and that for any given buffer size, longer periods of congestion result in an increase in cost. For instance, we can see that with a buffer size of 50 capsules, doubling the delay increases the cost 5.7 times and tripling the delay increases the cost 14.3 times. This is expected since the cost function grows in polynomial time.

Figure 5-1 Regular method buffer size vs. cost
Table 5-2 Active method costs

<table>
<thead>
<tr>
<th>COST</th>
<th>25</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 capsule delay</td>
<td>422.4 +/-31</td>
<td>111.6 +/-13</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 capsule delay</td>
<td>2922.4 +/-14</td>
<td>960.2 +/-213</td>
<td>240.6 +/-3</td>
<td>75 +/-0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300 capsule delay</td>
<td>7662 +/-39</td>
<td>4144.2 +/-1037</td>
<td>798.6 +/-14</td>
<td>379.2 +/-39</td>
<td>161.4 +/-2</td>
<td>74.4 +/-4</td>
<td>0</td>
</tr>
</tbody>
</table>

“Active” Method

The active method was simulated in the same way as the regular method. The results of the simulations with three nodes are given in Table 5-2. They are very similar to the regular method except that the costs are much lower across the board. A chart of these results is shown in Figure 5-2. Comparing this with Figure 5-1 we see that the trend lines are also very similar. Cost goes to zero as the buffer size equals the amount of congestion.

Figure 5-2 Active method size vs. cost
Figure 5-3 Percentage cost decrease--regular to active method

Figure 5-3 and Figure 5-4 compare the regular method to the active method. The percentage difference in cost at each (buffer size, delay) point was calculated. The active method has the smallest advantage when the buffer size is small and the delay is long. This is still a 22% decrease in cost. The difference between the two methods occurs while the active buffer fills up. Once it is filled with I-frame packets, it behaves the same as the regular method and drops all subsequent packets. At that point, the cost increases at the same rate as the regular method. The advantage for the active method grows quickly as the buffer size increases. With a larger buffer, there are more GOPs to select from, and the result is lower cost.
The active method and the dispersed method were both simulated using a (12, 3) GOP and a (18, 6) GOP with buffer sizes of 100 and 400 packets. The delay was varied up to the point at which the buffer no longer contains any B-frames. At that point, both methods act the same. Figure 5-5 and Figure 5-6 show the cost with a (12, 3) GOP. The dispersed method has a very slight advantage with a 100 packet buffer, and a small advantage with a 400 packet buffer when delays are short. The percentage of B-frames within this GOP is relatively small (66.6%), so the dispersion has fewer frames across which to be effective. Also, a total loss of B-frames between two I- or P-frames results in
the loss of just two consecutive frames. The cost of this is just 3. Since the idea of the dispersed method is to reduce the cost associated with consecutive B-frames lost, it is not surprising that the advantage of this method in this case is very small. Figure 5-6 shows that as the delay increases, the advantage of the dispersed method is completely nullified.
Figure 5-7 (18, 6) Active vs. dispersed, buffer = 100

Figure 5-7 and Figure 5-8 show the cost with a (18, 6) GOP. The advantage of the dispersed method is much greater in both cases. With a 100 packet buffer, the increased percentage of B-frames within each GOP (83.3%) combined with five consecutive B-frames between two I- or P-frames results in better performance for the dispersed method. With a 400 packet buffer, the improved results are even more apparent.
I-Frame Packet Density

The number of I-frame packets per GOP is termed the I-frame density. All simulations to this point have assumed 8 I-frame packets per GOP. A (12, 3) GOP would thus have an I-frame density of 32%. A (18, 6) GOP has an I-frame density of 28%. When the number of packets in I-frame is doubled, the I-frame density becomes 43%. This reduces the portion of B-frames that can be stored in the buffer. For the active method, this makes little difference in the final cost. Figure 5-9 shows that at small delays there is some benefit to having a larger I-frame, but as the delay grows there is no benefit. The effect of the larger I-frame is to shift the portion of discarded B-frames from the incoming packets to the packets in the buffer. Since either set of B-frame packets are discarded in the same manner (sequentially), the effect on cost is insignificant.

Figure 5-10 shows that when using the dispersed method there is a significant advantage to having a larger I-frame. The average improvement is 17.5%. There are fewer B-frames in the buffer because of the larger I-frame. The amount of dispersion is
therefore somewhat limited. However, this is more than offset by fewer B-frames being discarded when they try to enter the buffer and are rejected. Any B-frame that is received when the buffer is already full is discarded. Without knowing the congestion time in advance, there is no intelligent way to discard these frames. Thus, these frames are discarded in sequence and accumulate a higher cost than those discarded within the buffer, which are dispersed. Because the percentage of B-frames within a GOP is lower when the I-frame is larger, a larger portion of B-frames are discarded from the buffer than from the incoming packet stream. The cost of B-frames discarded from the buffer is lower when using the dispersed method, therefore the overall cost is lower.

Figure 5-10 (18, 6) I-frame size comparison, dispersed
CHAPTER 6
CONCLUSION

As expected, the regular method, which simulated a typical network router, incurred the highest cost when presented with network delays. The regular method’s naive way of dropping consecutive frames highlighted the cost function’s adverse reaction to long periods in which no frames are present. The active method showed marked improvement over the regular method, even with relatively small buffers. As buffer size increased, the improvement was dramatic. The algorithm is relatively simple and easy to implement within the ANTS framework. Placing the processing directly in the network node allowed the system to react to congestion immediately.

The dispersed method is an attempt to refine the active method’s simplified handling of B-frames. The advantage for small buffers was non-existent. Implementing the dispersed method in that case would not be recommended since quite a bit of additional processing is required. This method is worth the extra processing time when using larger buffers and GOPs with large numbers of B-frames. In that situation, the dispersed method showed real, consistent improvement over the active method. The density of I-frame packets within a GOP was shown to be immaterial in determining cost when using the active method. However, the dispersed method takes advantage of the lower percentage of B-frames within the GOP as the number of I-frames is increased, and shows real improvement as the I-frame density increases.
This thesis concentrated on implementing frame dropping algorithms at the packet level within the ANTS framework and using an appropriate cost function to assess those algorithms. No actual MPEG-2 video was encoded and sent over our simulated network. It would be valuable to do this and evaluate the resulting video subjectively, comparing the subjective evaluations to the cost function. Network congestion was simulated. It would also be valuable to have the system react to actual congestion, possibly by sending multiple MPEG-2 streams over the same network at the same time.
BIBLIOGRAPHY


Number of frames lost based on the number of packets lost and the starting point of the loss within the GOP. Figures are for a (12, 3) GOP.
APPENDIX B
CAPSULE CLASS

/*
 * MpegCapsule.Java
 * */

package apps.mpeg;

import ants.core.Protocol;
import ants.core.DataCapsule;
import ants.core.Xdr;
import ants.core.Node;
import ants.core.ByteArray;

/**
 * Mpeg capsule.
 *
 */
public class MpegCapsule
extends DataCapsule
{
    final private static byte[] MID = findMID("apps.mpeg" + ".MpegCapsule");
    final private static byte[] PID = findPID("apps.mpeg" + ".MpegCapsule");

    protected byte[] mid()
    {
        return MID;
    }

    protected byte[] pid()
    {
        return PID;
    }

    /**
     * Sequence number for the Mpeg capsule.
     */
    public int type;  //Capsule type 1=I-frame, 2=P-frame, 3=B-frame
    public int frameId;        //per frame sequence number
    public int seqId;  //intra-frame sequence number
    public int dest;  //the capsules final destination
public int length()
{
    int l = Xdr.INT + Xdr.INT + Xdr.INT + Xdr.INT;
    return super.length() + l;
}

difficult Xdr encode()
{
    Xdr xdr = super.encode();
    xdr.PUT(type);
    xdr.PUT(frameId);
    xdr.PUT(seqId);
    xdr.PUT(dest);
    return xdr;
}

difficult Xdr decode()
{
    Xdr xdr = super.decode();
    type = xdr.INT();
    frameId = xdr.INT();
    seqId = xdr.INT();
    dest = xdr.INT();
    return xdr;
}

public boolean evaluate(Node n)
/*
 * We want to deliver this to all except the first one
 * The node will decide whether to forward capsule or not
 */
{
    if ( n.getAddress() != getSrc() )
    {
        return n.deliverToApp(this,dpt);
    }
    else {
        return n.routeForNode(this, getDst());
    }
}

difficult MpegCapsule()
{
}

difficult MpegCapsule(short sa, short da, int na, ByteArray d)
{
    super(sa, da, na, d);
}
package apps.mpeg;

public class MpegProtocol
    extends ants.core.Protocol
{
    public MpegProtocol()
        throws Exception
    {
        startProtocolDefn();

        startGroupDefn();
        addCapsule("apps.mpeg" + ".MpegCapsule");
        endGroupDefn();

        endProtocolDefn();
    }
}
There are three parts to this appendix. The first part contains the code for the regular version of the application class. The second part contains the changes that were necessary to make the application class “active.” The third part contains the changes that were necessary to make the application disperse the B-frames.

```java
/*
 * MpegApplication.Java
 *
 * This is the "regular" version
 *
 */

package apps.mpeg;

import ants.core.Application;
import ants.core.Capsule;
import ants.core.Protocol;
import ants.core.Node;
import ants.core.NodeAddress;
import ants.core.Xdr;
import ants.core.ByteArray;
import ants.util.KeyArgs;
import ants.util.InvalidKeyArgsException;
import java.util.*;

/**
 * Takes most parameters via command line arguments.
 * Accepts the following parameters (in addition to the basic ones accepted by every Application).
 *
 * -target <addr>: Address for mpeg capsules to go to.
 * -arouter <addr>: Address of the active router.
 * -iter <int>: Numer of mpeg capsules to send [200].
 * -interv <int>: Inter-capsule delay in ms [18].
 * -prime <int>: Post prime packet delay. Neg means no prime packet. [-1].
*/
```
public class MpegApplication
extends Application
implements Runnable
{
    private final static String[] defaults = {};
    private final int LOG_LEVEL = L[6];
    private int prime = -1;
    private int target;
    private int arouter;
    private int rTotCount = 0;
    private int count = -1;
    private long beginTime;
    private int iter = 200;
    private int lossage = 10;
    private int interv = 18;
    private int bufferSize = 56;
    private int delay = 112;
    private int N = 12;
    private int M = 3;
    private int IPackets = 8;
    private int qCount = 0;  //counter for queue object
    RandomIntGenerator r1 = new RandomIntGenerator(25, 49);
    private int dstart = r1.draw();

    public void run()
    {
        int where = thisNode().getAddress();
        thisNode().log(LOG_LEVEL, "Waiting to receive packets. This address is: "+where);
    }
}
/Create an MpegReceiver
MpegReceiver(MpegApplication papp)
{
    this.papp = papp;
}

;/*
 * MpegRouter class
 * This class will implement the active router function in
 * this network
 */
class MpegRouter
implements Runnable
{
    MpegApplication rapp;
    private final int LOG_LEVEL = 1;

    public void run()
    {
        int where = thisNode().getAddress();
        thisNode().log(LOG_LEVEL, "This is the routing node. The address is: "+where);
    }

    //Create an MpegRouter
    MpegRouter (MpegApplication rapp)
    {
        this.rapp = rapp;
    }
}

/*
 * This is a class of objects to fill the buffer on each node
 * Capsules cannot be queued directly, so their contents must be
 * copied to a QueueObj, which is actually what is put into the
 * linked list
 */
class QueueObj
{
    private int type;
    private int frameID;
    private int seqID;
    private int dst;

    QueueObj (int type, int frameID, int seqID, int dst) {
        this.type = type;
        this.frameID = frameID;
        this.seqID = seqID;
        this.dst = dst;
        count++;
    }
private int getType() {
    return this.type;
}

private int getFrameID() {
    return this.frameID;
}

private int getSeqID() {
    return this.seqID;
}

/*
 * Invoked when packets are received.
 */
synchronized public void receive(Capsule unknownCap) {
    long now;
    super.receive(unknownCap);
    if (!(unknownCap instanceof MpegCapsule)) {
        thisNode().log(LOG_LEVEL, "Non-Mpeg capsule delivered?!!" + unknownCap);
        thisNode().log(LOG_LEVEL, "MpegCap.classLoader= " + MpegCapsule.class.getClassLoader() + " other=" + unknownCap.getClass().getClassLoader());
        return;
    }
    MpegCapsule cap = (MpegCapsule) unknownCap;
    now = thisNode().time();
    int where = thisNode().getAddress(); // the current node
    /* Here we need to split the receive method into two parts
     * One part is for the router node the other is for the
     * receiver
     */
    if (cap.dest != where) {
        // Get the capsules header information
        int a = cap.type;
        int b = cap.frameId;
        int c = cap.seqId;
        int d = cap.dest;
// A bunch of stuff to translate the frame type
String t2 = "?";
switch(a)
{
    case 1:  t2 = "I";
        break;
    case 2:  t2 = "P";
        break;
    case 3:  t2 = "B";
        break;
    default: t2 = "??";
        break;
}

QueueObj q = new QueueObj (a, b, c, d);
// dstart is a random number between 25 and 49 (unless
// specified in mpeg.config)
// so that the start point for delay is random
// If during congestion, queue the capsule while queue not
// full
if ( ( count>=dstart) && (count < dstart +
    bufferSize) && (count-dstart < delay) )
{
    queue.add(q);  //add to end of queue
    thisNode().log(LOG_LEVEL, "Queueing
        ("+t2+","+b+","+c+"));
    qCount++;
}

// Congestion is now over so we can send from the queue
// The number in the array is the smaller of bufferSize and
// delay
if (count == (dstart + delay) )
{
    QueueObj temp;
    // Iterator e = queue.iterator();
    while(!queue.isEmpty())
    {
        temp = (QueueObj)queue.removeFirst();
        // A bunch of stuff to translate the frame type
        int t3 = temp.type;
        String t4 = "?";
        switch(t3)
        {
            case 1:  t4 = "I";
                break;
            case 2:  t4 = "P";
                break;
            case 3:  t4 = "B";
                break;
            default: t4 = "??";
                break;
        }
    }
}
sendCapsule(t3, temp.frameID, temp.seqID);
thisNode().log(LOG_LEVEL, "Forwarding
("+t4+","+temp.frameID+","+temp.seqID+")
from buffer.");
}

else   // this is the receiver
{

//A bunch of stuff to translate the frame type
int t1 = cap.type;
String t2 = "?";
switch(t1)
{
    case 1:   t2 = "I";
             break;
    case 2:   t2 = "P";
             break;
    case 3:   t2 = "B";
             break;
    default:  t2 = "??";
             break;
}

thisNode().log(LOG_LEVEL, "MpegCapsule
("+t2+","+cap.frameId+","+
cap.seqId + ")"
+ " " (+rTotCount+ " received");

if (cap.seqId == -1) // don't count the prime capsule
    return;

rTotCount++;

if (rTotCount >= iter - lossage)
{
    thisNode().log(LOG_LEVEL, "Received almost all
capsules. Exiting.");
    thisNode().shutdown();
}

else // this is the receiver
{

    // else send the capsule if it is before or after the
    // congestion
    // capsules not queued or sent from here will be dropped
    if ( (count >= (dstart + delay)) || (count < dstart) )
    {
        sendCapsule(cap.type, cap.frameId, cap.seqId);
        thisNode().log(LOG_LEVEL, "Forwarding
("+t2+","+cap.frameId+","+cap.seqId+");
    }
    return;
}
public void run()
{
    int t = 1;
    int f = 2;
    int i = 1;
    int p = 1;
    int b = 1;
    int where = thisNode().getAddress();
    thisNode().log(LOG_LEVEL, "sending "+iter+" capsules at "+
    +interv+"ms intervals from: "+
    +where);

    try
    {
        // First, send a warmup capsule to get the protocols loaded at
        // each node.
        if (prime > 0)
        {
            thisNode().log(LOG_LEVEL, "Sending primer capsule (and waiting "+
            +prime+"ms)." );
            sendCapsule(1, 1, -1);
            thisNode().sleep(prime);
        }

        thisNode().log(LOG_LEVEL,"Sending Mpeg capsules from:
        "+thisNode().getAddress());
    }

    /*
     * A rather complicated set of stuff to send a complete MPEG
     * frame
     * The number of frames sent is k/25
     */
    int numP = (N / M) - 1; //number of P-frames per GOP
    int numB = N - numP - 1; //number of B-frames per GOP
    int Bper = numB / (numP + 1); //number of B-frames in
    //between I/P and P frames
    int bCount = 0; //label for the B-frames within a GOP
    int pCount = 0; //label for the P-frames within a GOP
    for (int k=0; k < (int)iter/25; k++)
    {
        bCount = 0;
        pCount = 0;
        for (i=1; i<=IPackets; i++)
        {
            t = 1;
            thisNode().log(LOG_LEVEL,"Sending Mpeg capsule
            
            (and waiting "+
            +prime+"ms)." );
            sendCapsule(1, 1, -1);
            thisNode().sleep(prime);
        }
    }
+t","+ f","+i") from:
"+thisNode().getAddress());
sendCapsule(t, f, 1);
thisNode().sleep(interv);
}
for (b=(numB - Bper + 1); b<= numB; b++)  {
    t = 3;
    thisNode().log(LOG_LEVEL,"Sending Mpeg
capsule("+t","+ (f-1) "+","+b") from:
"+thisNode().getAddress());
sendCapsule(t, (f-1), b);
thisNode().sleep(interv);
}
for (int repeat = 1; repeat <= numP; repeat++)  {
    pCount++;
    for (p=1; p<=3; p++)  {
        t = 2;
        thisNode().log(LOG_LEVEL,"Sending Mpeg
capsule ("+t","+ f","+pCount+")
from:"+thisNode().getAddress());
sendCapsule(t, f, pCount);
thisNode().sleep(interv);
    }
    for (b=1; b<=Bper; b++)  {
        t = 3;
        bCount++;
        thisNode().log(LOG_LEVEL,"Sending Mpeg
capsule ("+t","+ f","+bCount+") from:
"+thisNode().getAddress());
sendCapsule(t, f, bCount);
thisNode().sleep(interv);
    }
}
    f++;
}
thisNode().log(LOG_LEVEL, "Sent all capsules.
Press <Ctrl-C> to exit.");
thisNode().sleep(10000);
//thisNode().shutdown();
}
catch (InterruptedException _)
{
    thisNode().log(LOG_LEVEL, "MpegApp Interrupted.
    Try again.");
}

private void sendCapsule(int t, int f, int k)
{
    ByteArray buf = new ByteArray(Xdr.LONG);
Xdr xdr = new Xdr(buf, 0);
xdr.PUT(thisNode().time());
MpegCapsule c = new MpegCapsule(getPort(), getPort(),
    target, buf);
c.dest = c.getDst();
c.type = t;
c.frameId = f;
c.seqId = k;
send(c);
}
/**
 * Parse command line arguments.
 * See the class description for a list of the arguments and
 * their meaning.
 */
public void setArgs(KeyArgs k)
throws InvalidKeyArgsException
{
    k.merge(defaults);
    try
    {
        for (int i = 0; i < k.length(); i++)
        {
            if (k.key(i).equals("-target"))
            {
                target = NodeAddress.fromString(k.arg(i));
                k.strike(i);
            }
            if (k.key(i).equals("-arouter"))
            {
                arouter = NodeAddress.fromString(k.arg(i));
                k.strike(i);
            }
            if (k.key(i).equals("-dstart"))
            {
                dstart = Integer.parseInt(k.arg(i));
                k.strike(i);
            }
            if (k.key(i).equals("-bufferSize"))
            {
                bufferSize = Integer.parseInt(k.arg(i));
                k.strike(i);
            }
            if (k.key(i).equals("-delay"))
            {
                delay = Integer.parseInt(k.arg(i));
                k.strike(i);
            }
            if (k.key(i).equals("-iter"))
            {
iter = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-interv"))
{
    interv = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-lossage"))
{
    lossage = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-prime"))
{
    prime = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-N"))
{
    N = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-M"))
{
    M = Integer.parseInt(k.arg(i));
k.strike(i);
}
if (k.key(i).equals("-IPackets"))
{
    IPackets = Integer.parseInt(k.arg(i));
k.strike(i);
}
}
}

catch (Exception e)
{
    throw new InvalidKeyArgsException("HACK HACK: ",
    +e.getMessage());
}

super.setArgs(k);

/**
 * Start the Non-GUI Mpeg app.
 *
 */

public void start()
throws Exception
{
    thisNode().register(new MpegProtocol());
if ( (target != thisNode().getAddress()) && (arouter !=
    thisNode().getAddress()) )
{
    //this is the source node
    thisNode().threadStart(this);
}
else if (arouter == thisNode().getAddress())
{
    //this is the active router
    thisNode().threadStart(new MpegRouter(this));
}
else
{
    //this is the target node
    thisNode().threadStart(new MpegReceiver (this));
}

/**
 * Construct a non-GUI Mpeg app.
 * Doesn't do anything.
 */
public MpegApplication()
throws Exception
{
}
}
In order to make the application class “active,” substantial changes were made to the receive method. In addition, a method was added to help with removal of P-frames.

```java
/**
 * The receive() method is invoked when packets are received.
 */
synchronized public void receive(Capsule unknownCap)
{
    long now;
    super.receive(unknownCap);
    if (!(unknownCap instanceof MpegCapsule))
    {
        thisNode().log(LOG_LEVEL, "Non-Mpeg capsule delivered?!!" +unknownCap);
        thisNode().log(LOG_LEVEL, "MpegCap.classLoader= "
            +MpegCapsule.class.getClassLoader() +
            "; other="
            +unknownCap.getClass().getClassLoader());
        return;
    }
    MpegCapsule cap = (MpegCapsule)unknownCap;
    now = thisNode().time();
    int where = thisNode().getAddress();
    //Used to debug what the current node is
    /* At this point we need to split receive method in two parts
     * One part is for router node the other is for the receiver
     */
    //If this is not the destination then this is a router
    if (cap.dest != where)
    {
        // Get the capsules header information
        int a = cap.type;
        int b = cap.frameId;
        int c = cap.seqId;
        int d = cap.dest;
        boolean added2Q = false;
        ListIterator e = (ListIterator)queue.iterator();

        //A bunch of stuff to translate the frame type
        String t2 = "?";
        switch(a)
        {
            case 1:    t2 = "I";
                        break;
            case 2:    t2 = "P";
```
break;
case 3:  t2 = "B";
    break;
default: t2 = "??";
    break;
}

QueueObj q = new QueueObj (a, b, c, d);
// dstart is a random number between 25 and 49 (unless
// specified in mpeg.config)
// so that the start point for delay is random
//If during congestion, queue the capsule while queue not full
if ( (count>=dstart) && (count < dstart + delay) &&
    (count-dstart < delay) )
{
    // the queue is not full so add the incoming packet
    // to it if it is not a B-frame
    if( (queue.size() < bufferSize) && (a != 3) ) {
        queue.add(q);
        thisNode().log(LOG_LEVEL, "Queueing 
    
    "+t2+","+b+","+c+");
        a = 3;//will get us out the switch stmt fast
    }
    // If is a B-frame packet we only add to the queue
    // if it has never been full
    else if( (queue.size() < bufferSize) &&
    (!fullOnce) ) {
        queue.add(q);
        thisNode().log(LOG_LEVEL,"Queueing 
    
    (B,"+b+","+c+");
        a = 3;
    }
    switch (a)
    {
    case 1:  //Incoming packet is an I-frame
    {
        fullOnce = true;
        resetIterator(e);
        while (e.hasNext()){//iterate queue and
            QueueObj r = (QueueObj)e.next();
            //if this is a B-frame drop it and enqueue the I packet
            if (r.getType() == 3) {
                e.remove();
                queue.add(q);
                thisNode().log(LOG_LEVEL, 
            
            "Queueing (I,"+b+","+c+");
            added2Q = true;
            break;
            }
            if(added2Q)
                break;
    }
The next three iterations drop P-frame packets selectively highest subscript first

```c
resetIterator(e);
while (e.hasNext()) {
    // iterate the queue and
    QueueObj r = (QueueObj)e.next();
    // if this is a P-frame check the subscript and if it is 3 . . .
    if( (r.getType() == 2) &&
        (r.getSeqID() == 3) ) {
        removeFrame(e.previousIndex());
        // remove the P-frame
        queue.add(q);
        // add the incoming I packet to the queue
        thisNode().log(LOG_LEVEL, "Queueing (I,"+b+","+c+")");
        added2Q = true;
        break;
    }
    if(added2Q)
        break;
    resetIterator(e);
    while (e.hasNext()) {
        // iterate the queue and
        QueueObj r = (QueueObj)e.next();
        // if this is a P-frame check the subscript and if it is 2 . . .
        if( (r.getType() == 2) &&
            (r.getSeqID() == 2) ) {
            removeFrame(e.previousIndex());
            // remove the P-frame
            queue.add(q);
            // add the incoming I packet to the queue
            thisNode().log(LOG_LEVEL, "Queueing (I,"+b+","+c+")");
            added2Q = true;
            break;
        }
        if(added2Q)
            break;
        resetIterator(e);
        while (e.hasNext()) {
            // iterate the queue and
            QueueObj r = (QueueObj)e.next();
            // if this is a P-frame check the subscript and if it is 1 . . .
            if( (r.getType() == 2) && (r.getSeqID() == 1) ) {
                removeFrame(e.previousIndex());
            }
        }
    }
```
// remove the P-frame
queue.add(q);

//add the incoming I packet to the queue
thisNode().log(LOG_LEVEL,"Queueing
(I,"+b","+c")");
added2Q = true;
break;

// otherwise the queue contains only I-frames so we drop the
// incoming I packet
break;

case 2: //Incoming packet is a P-frame packet
{
    fullOnce = true;
    resetIterator(e);
    while (e.hasNext()) {
        // iterate the queue and
        QueueObj r = (QueueObj)e.next();
        if (r.getType() == 3) {
            // if this is a B-frame drop it and enqueue the P packet
            e.remove();
            queue.add(q);
            thisNode().log(LOG_LEVEL,"Queueing
(P,"+b","+c")");
            added2Q = true;
            break;
        }
    }
    if(added2Q)
    break;
    resetIterator(e);
    while (e.hasNext()) {
        // iterate the queue and
        QueueObj s = (QueueObj)e.next();
        // if this is a P-frame check the subscript
        // and if incoming is less than queued ...
        if( (s.getType() == 2) && (c <
            s.getSeqID()) ) {
            removeFrame(e.previousIndex());
        }
    }
    // remove the P-frame
    queue.add(q);
    //add the incoming P packet to the queue
    thisNode().log(LOG_LEVEL,"Queueing
    (P,"+b","+c")");
    added2Q = true;
    break;
}

//otherwise the queue contains only I-frames so
//we drop the incoming P packet
break;
}
case 3:
// Incoming packet is a B-frame ... drop it
{
    break;
}
}
qCount++;

// Congestion is now over so we can send from the queue
// The number in the array is the smaller of bufferSize and delay
if (count == (dstart + delay))
{
    QueueObj temp;
    while(!queue.isEmpty())
    {
        temp = (QueueObj)queue.removeFirst();
    }

    // A bunch of stuff to translate the frame type
    int t3 = temp.type;
    String t4 = "?";
    switch(t3)
    {
        case 1: t4 = "I";
            break;
        case 2: t4 = "P";
            break;
        case 3: t4 = "B";
            break;
        default: t4 = "??";
            break;
    }
    sendCapsule(temp.type, temp.frameID,
                temp.seqID);
    thisNode().log(LOG_LEVEL, "Forwarding
           ("+t4+","+temp.frameID+","+temp.seqID+") from
           buffer.");
}

// else send the capsule if it is before or after the congestion
// capsules not queued or sent from here will be dropped
if ( (count >= (dstart + delay)) || (count < dstart) )
{
    sendCapsule(cap.type, cap.frameId, cap.seqId);
    thisNode().log(LOG_LEVEL, "Forwarding
           ("+t2+","+cap.frameId+","+cap.seqId+")");
}
return;
else  // This is the Receiver
{
int t1 = cap.type;
String t2 = "?";
switch(t1)
{
    case 1:    t2 = "I";
        break;
    case 2:    t2 = "P";
        break;
    case 3:    t2 = "B";
        break;
    default:   t2 = "??";
        break;
}
thisNode().log(LOG_LEVEL, "MpegCapsule" + t2 +"," + cap.frameId +"," + cap.seqId + ")" + " (" + rTotCount+ " received)");

if (cap.seqId == -1) // don't count the prime capsule
return;
rTotCount++;
if (rTotCount >= iter - lossage)
{
    thisNode().log(LOG_LEVEL, "Received almost all capsules. Exiting.");
    thisNode().shutdown();
}
}

private void removeFrame(int index) {
    for (int i = 1; i<=3; i++) {
        if (index == queue.size()) {
            return;
        }
        QueueObj q = (QueueObj)queue.get(index);
        if (q.getType() == 2) {
            queue.remove(index);
        }
    }
    return;
}
To facilitate the "dispersed" version of the application the “b smart” version contains two extra methods. The first one labels all the B-frames in the queue numerically so they can be removed in the next method in a way that is “smart.”

```java
/*
* Label the B-frames in the queue
* This should only happen once, the first time queue is full
* Iterate the queue and number B-frame packets sequentially
*/
private void labelQueue() {
    int i = 1;
    ListIterator e = (ListIterator)queue.iterator();
    resetIterator(e);
    while (e.hasNext()) {  // iterate the queue and ...
        QueueObj r = (QueueObj)e.next();
        if (r.getType() == 3) {  // if this is B-frame label it
            r.setLabel(i);
            i++;
        }
    }
    numLabels = i - 1;  // The total number of labels
    return;
}

/*
* This method puts the "smart" in the bsmart version.
* We iterate the queue and remove the B-frame packet that is
* the furthest from other previously removed B packets
*/
private void removeBFrame() {
    bCount++;
    if (bCount == 2) {
        multiplier = 1;  // special case adjustment
    }
    if (bCount > (numLabels/2)) {
        setFlag = true;
    }
    float num = (float) numLabels;
    float mult = (float) multiplier;
    float remove = (num / mult) * adder;
    int labelToRemove = (int)Math.ceil(remove);
    if (bCount == 1) {
        labelToRemove = 1;
        int check1;
        ListIterator e = (ListIterator)queue.iterator();
        resetIterator(e);
        while (e.hasNext()) {  // iterate the queue and
            QueueObj r = (QueueObj)e.next();
```
if (r.getType() == 3) { //if this is B-frame check

// the label
    check1 = r.getLabel();
    if(check1 == labelToRemove) {
        e.remove();
        thisNode().log(LOG_LEVEL,
            "(B,"+r.frameID+","+r.seqID+)
            removed ("+r.label")");
        break;
    }
}
}

else if( flag ) {
    // We are more than halfway through the B-frame packets so it
    // does not matter which one we remove -- remove whichever is
    // next
    ListIterator e = (ListIterator)queue.iterator();
    resetIterator(e);
    while (e.hasNext()) { //iterate the queue and
        QueueObj r = (QueueObj)e.next();
        if (r.getType() == 3) {
            //if this is a B-frame remove it
            e.remove();
            thisNode().log(LOG_LEVEL,
                "(B,"+r.frameID+","+r.seqID+)
                removed ("+r.label")");
            return;
        }
    }
}

else {

    /*
    * Now iterate the queue and remove B-frame packet with label
    * that matches labelToRemove
    */
    int check2;
    ListIterator e = (ListIterator)queue.iterator();
    resetIterator(e);
    while (e.hasNext()) { //iterate the queue and ...
        QueueObj r = (QueueObj)e.next();
        if (r.getType() == 3) {
            // if this is a B-frame check the label
            check2 = r.getLabel();
            if(check2 == labelToRemove) {
                e.remove();
                thisNode().log(LOG_LEVEL,
                    "(B,"+r.frameID+","+r.seqID+)
                    removed ("+r.label")");
            }
        }
    }
}
break;
}
else if(check2 > labelToRemove) {
    e.remove();
    thisNode().log(LOG_LEVEL,
        "(B,"+r.frameID+","+r.seqID+"
        ) removed ( "+r.label+" )");
    break;
}

/*
 * Now we set the variables to the values needed at the start
 * of the next iteration of this method
 */
if( (adder+2) < multiplier) {
    adder = adder + 2;
} else {
    if(setFlag) {
        flag = true;
    }
    multiplier = multiplier * 2;
    adder = 1;
}
return;
BIOGRAPHICAL SKETCH

The author was born in Henderson, KY, and spent most of his childhood growing up in Indiana and Michigan. In 1981, his family moved to Florida where he began high school. In May 1990, he earned a Bachelor of Science in Business Administration from the University of Florida. He held several positions after graduation, the most recent of which was as a franchisee of the Great American Cookie Company with a store in Ocala, FL. In the fall of 1999, he returned to the University of Florida and began his study of computer engineering.