An Examination of Wireless Media Streaming in a Home Network Environment

By

Jay Kraut

A Thesis

Submitted to the Faculty of Graduate Studies
In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba

Abstract

The amount of consumer electronics today has increased substantially. The average home might have multiple VCRs, DVDs, PVRs and display devices. Increasingly, there is a push towards building all in one devices as media servers and making them capable of streaming media to various display devices in the home. At the same time, wireless connectivity has improved considerably with the advent of the 802.11 standard.

The purpose of this thesis is to give a broad overview of how wireless media streaming works. The 802.11 standard is examined through a custom made wireless simulator. The development of media streaming software is achieved through using the Microsoft DirectShow architecture. This thesis explains how both the simulator and streaming software is developed, with an emphasis on software engineering issues. The design of the software is comprehensively explained with the addition of code samples in the main text to further the explanations.

The focus of this thesis is more on gaining an understanding of how media streaming works rather than focusing on detailed experimentation with both 802.11 and streaming media wirelessly. There are several test scenarios for both 802.11 using the wireless simulator and media streaming, however these are more of an example of what the software for this thesis can do rather then a scientific analysis.

This thesis should be useful for those interested in developing either wireless network simulators or video streaming applications. As well it will be useful for those who want a greater understanding of some of the issues discussed and perhaps some of the software engineering issues such as component based architectures.

Acknowledgments

I would like to thank the people who helped me along the way and made it possible to complete this thesis. First of all, I would like to thank my Advisor Prof. Bob McLeod for being a great advisor and for teaching an excellent course on wireless network simulators. I would like to thank TRLabs and NSERC for providing funding and again to TRLabs for providing a great work environment and all the equipment required for doing the testing. Much appreciation goes to Sergey Giterman for answering my many questions via one of the tech forums and later for replying to my emails when I was first learning about DirectShow. Also thank you to everyone in the Microsoft DirectShow video forum who offered advice. Lastly I would like to thank both my sister and my mother for doing the grammar check on the thesis.

Table of Contents

Chapter 4 DirectShow Filters 60

Chapter 5 Media Streaming Architecture 81

Chapter 6 Wireless Video Streaming Experiments 114

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem Context

There is an increasing amount of convergence in the home entertainment electronics industry. There are both technological and marketing reasons for this. Advances in compression technology, coupled with lower costs and standardization of integrated computing devices, has enabled the development of all-in-one devices. On the demand front, with the average home having multiple media devices such as DVDs, computers, gaming consoles and stereos, there is an increasing need to have them all interconnect seamlessly. Currently, there are many manufactures such as Sony, Samsung and Creative that are manufacturing devices that are multifunctional. However, these devices are still relatively immature and it is uncertain what the home electronics market will look like in the future.

One thing that is certain to happen in the future, is an increase in the reliance on wireless networks instead of wired ones. Wireless networks are slower but have advantages. Wireless networks eliminate much of the clutter of connecting an increasing number of devices with wires. Also, wireless networks provide added mobility, which is practical for devices such as laptops, and other portable media devices.

The streaming of media (both audio and video) to one or many wireless displays is becoming more common. If bandwidth is unlimited, and bit error rate negligible then reliable streaming is relatively easy to implement. However, as the bit rate increases with higher quality video formats for example HDTV maintaining a high Quality of Service

(QoS) becomes difficult. This project looks at issues with streaming media over a wireless home network.

1.2 Scope

This thesis seeks to answer the following research question: Fundamentally, how does wireless media streaming work? The context of this question is in the application of streaming media in a home network environment. That is, there is a media server in a home that is streaming media through an access point to video and audio displays that are connected wirelessly via the access point.

The thesis is restricted to current technologies. It is possible to increase the efficiency of a wireless standard by customizing it to specific applications. However hardware implementation of a new wireless standard is quite difficult and beyond the scope of this thesis. This means that the wireless devices used for this thesis are restricted to ones available as this thesis is written (802.11 a,b,g). However there is an explanation on how wireless works from the media access control (MAC) layer and up this is done to examine any MAC issues that may affect how to program in the transport layer effectively.

To compensate for the use of pre-existing standards and to give greater insight into how a wireless network affects media streaming, a modifiable wireless simulator is developed. The wireless simulator is made to reproduce the results of experimentation with streaming software to give a greater insight into what is happening under heavy network traffic. It also has the capability to simulate multiple channels to take a look future wireless standards. This simulator is “real time” and event based, similar to other simulators available today such as NS2 [NS204] and J-Sim [JSIM04]. It can run real applications over the network instead of simulated traffic, which makes the results more realistic [AhDa96] [FPJF03][HuKH04][Tyan02]. Other features are data integrity that allows a mock ftp program that correctly sends data over the network and that the simulator supports multiple transmission channels over a network. Even though other network simulators are available, the simulator is developed for this project because it is simple to use, easily modifiable and has provided a great learning experience.

The original goal of this thesis is to create a small custom wireless media streaming box. This box would be able to accept analog video and audio input, compress it, and then stream it to a second box. The second box would decompress the stream and output it to a video monitor and audio speakers. Early on it is found that to create a compact box that fits the home media image is unrealistic. This would involve doing a lot of work with prototyping hardware, which is too expensive and time consuming for this thesis. The first design decision is to use off the shelf hardware namely two personal computers (PCs). The second design decision is to use current software technologies. A full custom programmed media streaming application including encoders and decoders is unrealistic to be implemented as a master's thesis. This thesis instead takes advantage of existing software technologies from Microsoft and concentrates on implementing all of the transport layer functionality.

The goal of the thesis is not to create commercial quality streaming software. There are many software vendors that provide streaming solutions based on similar architectures to the one used in the thesis (e.g. [Micro04]). There are also free versions available under Linux that are somewhat different (e.g. [ViLa04]). The goal of the thesis is to provide a basic understanding of how wireless streaming works all the way from the software architecture level, down to the MAC level on the wireless network. The thesis provides a background in each subject and sample code that can be reapplied when developing media streaming devices. The media streaming software can also be used to test different hypothetical scenarios under different network conditions.

1.3 Outline

As much as possible, full explanations are provided starting from software architecture to detailed explanations of the actual code. Some sections in the thesis are written in tutorial style, many of them containing code snippets that are provided to further the understanding of the material in a given section. Care is also taken to explain software engineering issues especially those involving the simulator architecture with its easy to program API and the design of the media streaming application.

Chapter 2 introduces the wireless simulator. First, an overview of how 802.11 works is described in sections 2.1 and 2.2. Next, the simulator architecture is described in section 2.3. Section 2.4 discusses how to program applications for the simulator.

Chapter 3 discusses the simulation experimentation. It starts with some experimentation to verify that the simulator is working correctly which is presented in section 3.1, and which provides some sample code to show how simple it is to program the simulator. Section 3.2 contains several experiments. The experiments are not wholly intended to provide directly applicable results. The section has the dual purpose to show what can be done in the simulator and to show how it is possible for hardware developers to first test custom wireless standards to verify the properties before spending the time to implement them in hardware.

Chapter 4 starts the discussion on media streaming application software. Care is taken to introduce the Microsoft architecture in sections 4.1 to 4.4, which can be a difficult topic for novices. Section 4.5 further describes Directshow by illustrating a very common example.

Chapter 5 discusses the software architecture used to stream media. It is divided into three main sections. Section 5.1 discusses the transport layer, which is in both the server and the client and is the heart of this thesis. Section 5.2 discusses the server architecture, and section 5.3, the client architecture.

Chapter 6 describes the experiments done using the media streaming software. It starts by introducing one additional software utility to test a network. It follows with instructions on how to install the server and the client applications. Two hypothetical scenarios, one when the bandwidth of the network is completely used up and other with high packet loss rates, are evaluated using the media streaming software.

Chapter 7 provides a summary of the thesis and emphasizes some of the more important points.

Chapter 2 Wireless Simulation Background

2.1 CSMA-CD

Carrier Sense Multiple Access – Collision detection (CSMA-CD) is a protocol that allows multiple devices to share a single channel. It is the protocol used for the physical layer of 802.3 Local Area Network (LAN) connections when multiple devices are connected to the same transmission medium rather then hooked up to a hub or router.

The protocol is best explained by describing its operation by example. When a device has a packet to transmit, it monitors the transmission channel and waits until the channel is idle. When the channel becomes idle the device starts transmitting the packet on the channel. Once a device starts transmitting, if it detects that there is no other device attempting to transmit within two propagation delays it knows it has control of the channel and continues to transmit. After two propagation delays, it is guaranteed that all other devices on the network realize the device is transmitting and won't interfere. However, if another device starts transmitting before it detects the first device's transmission (this is possible within one propagation delay) there will be a collision. At this point both devices are listening to the channel and realize the data on the channel is not the same as what they are transmitting so both devices acknowledge that some other device is attempting to transmit. As soon as this occurs, both devices transmit a short jamming signal and cease to transmit. The devices then set a back off timer to wait before attempting to transmit again. How this back off time is generated is crucial to the proper operation of the network. If both devices are conFigured to immediately begin retransmitting right after a collision there will never be a time when both devices will be able to transmit. However, if the back off time is too large the network will be inefficient.

There are several attributes that a CSMA-CD network has. The first is a maximum bit rate. The maximum bit rate is the total number of bits that can be transmitted on a channel per a time period. This is the maximum theoretically possible rate and is not achievable if more then one device is sharing a channel. Another important attribute is the minislot time. This time is specified as twice the propagation delay. The minislot time determines the smallest packet size. If a packet is smaller then the minislot time there is a possibility that even if there is a collision, the transmitting device will assume the transmission was successful because it won't be able to detect the collision. Lastly, the network has a maximum packet size and a back off algorithm, both of which can be changed to maximize efficiency. An example of a back off algorithm that is commonly used is the $2^k – 1$ back off algorithm. The back off time after a collision is determined by taking a random number between 0 and $2^k – 1$ where k is the number of retransmitting attempts, limited to a maximum of 10.

2.2 CSMA-CA

There are many differences between the protocols for wireless and wired network connection. Wireless transmission provides a less a controlled environment then wired transmission. 802.11 defines the wireless transmission standard [Bren97] [IEEE99] [Nede01]. It contains provisions for specifying the physical transmission, authentication, encryption, power mode, roaming capabilities and other characteristics. The scope of this project only focuses on the transmission of data in a unchanging environment, where the number of stations is fixed and an access point exists.

In a wired LAN, CSMA-CD is used as the protocol. It is a good choice giving good efficiency especially for large data and being fair in terms of letting multiple devices have access to the channel. However, for a wireless environment it is not suitable for the following reasons:

1) The hidden station problem. This is illustrated by Figure 2-1. If station A transmits, station B is able to listen to the transmission but station C cannot because it is out of range. Because of this, station C can transmit if it has data to send and thus interferes with stations A's transmission.

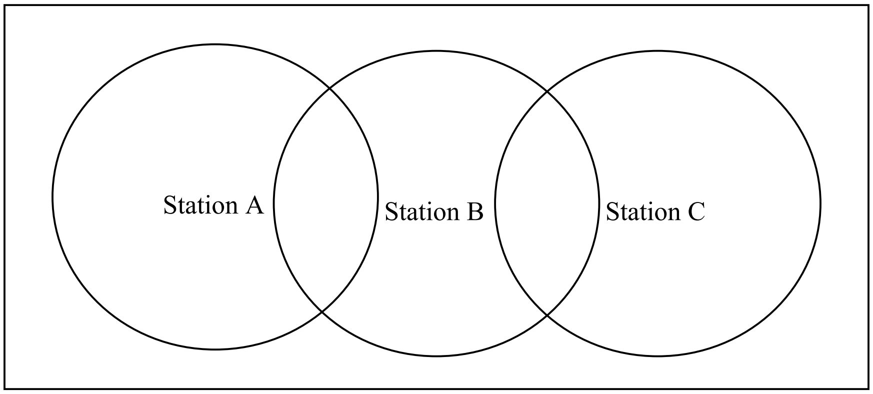


Figure 2-1 Hidden station problem

2) It is difficult to detect collisions in a radio environment. Also it is more expensive to build a device that can both transmit and receive back the single so it can check the integrity of the transmission. It may not be possible for the transmitter device to notice an error and stop transmitting.

3) For a given LAN setup, a station may not have complete control of the channel. There may be other devices on the same channel forming another LAN.

Because of these problems wireless Ethernet uses CSMA-CA, which is Carrier-Sense Multiple Access with Collision Avoidance. To coordinate its operation, CSMA-CA defines four kinds of Inter Frame Spaces. An Inter Frame Space (IFS) specifies the time between two transmissions. It is required to prioritize transmissions:

1) SIFS, Short interframe space. This is the smallest interframe space. It is necessary after a transmission for the wireless device to process the data and switch modes between receiving and transmitting if necessary. This actual time for 802.11 is around 28 uS.
2) PIFS, Point Coordinate IFS. This is the time after which the access point is able to get access to the channel without having to compete with all the other devices. This can be used to do access point functionality such as sending out beacons. It is defined as SIFS + 1 Slot time which is around 28+50 = 78 uS for 802.11.
3) DIFS, Distributed IFS is the minimum time after a transmission of a message which the stations can again start transmitting messages. It is defined as PIFS + 1 slot time = 132 uS.
4) EIFS, Extended IFS, is used when a carrier has received a packet it does not understand and should be kept from transmitting. EIFS is not used in any of the discussions in this thesis.

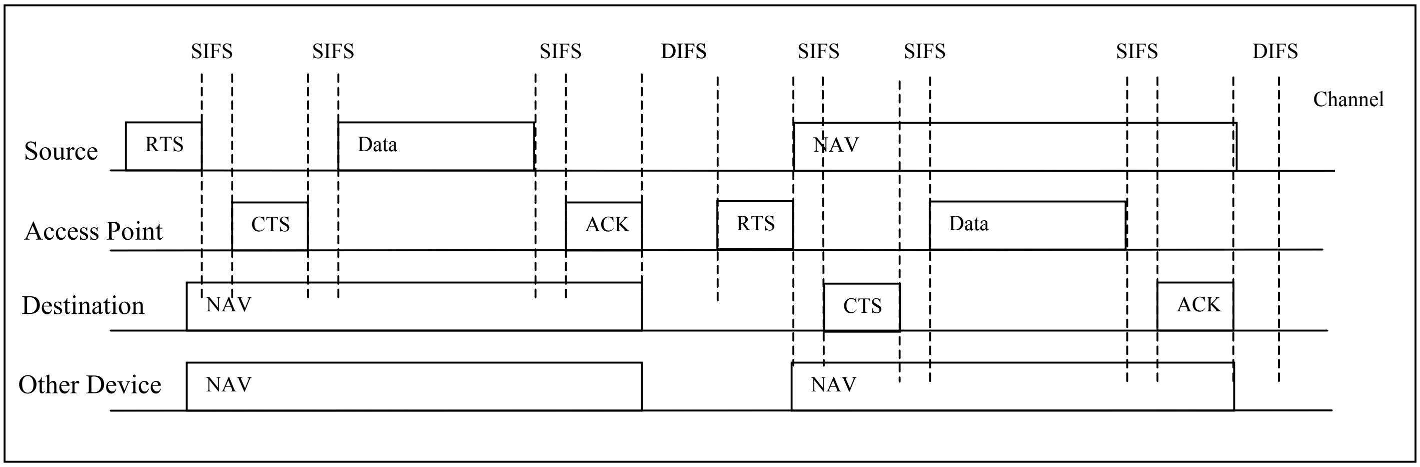


Figure 2-2 CDMA-CA example

The operation of CSMA-CA is illustrated in Figure 2-2. It demonstrates the procedure to transmit a packet from a source to a destination going through the access point. A transmission begins when a source sensing that the channel is idle after having its back off timer run to zero. It starts by sending a Request to Send (RTS) frame. If no other station is transmitting, the access point may successfully receive the frame. At this point the access point sends out a Clear to Send (CTS). Once received the source sends out the data frame. The access point acknowledges it with an acknowledgment (ACK) frame, and the transmission of the data frame is confirmed as being successful. Next the access point waits for the next available time to send and then using the same procedure sends the data frame to the destination.

CSMA-CA uses virtual carrier sensing. In the RTS and CTS frames, the time that is required to complete the transmission to the ACK frame is included in the header as the

Network Allocation Vector (NAV). This solves the hidden station problem. That station might not be able to detect the RTS or any subsequent transmission from the source, however, it will detect the CTS frame. At this point it reads the NAV and will not try to send until this time runs out.

The example in Figure 2-2 illustrates a transmission between two stations using an access point. This style of transmission requires station-to-station traffic to be sent twice which is inefficient. It is also possible for an ad-hoc network to be created which operates without the need for an access point. In such a network transmissions are sent peer to peer without having to be sent twice. Despite station-to-station traffic having to be sent twice, the access point scenario is the most common. The access point is also likely to be connected to the wired network with Internet and server connections. In many environments most of the traffic on the network is likely to go from access point to a station or vice versa rather then station to station.

Like CSMA-CD, CSMA-CA contains a back off algorithm. It is very similar and the formula to determine the time is given by 2^{k+2}. There is a maximum of 16 resend attempts before a packet is dropped.

It is possible for a wireless network to be conFigured with an RTS threshold. The RTS threshold is a size at which, if the data packet is smaller, the data packet can be sent without the RTS, CTS overhead. The reason behind the RTS and CTS is due to the fact that during a transmission, it is not possible to detect collisions and bit errors. When the RTS and CTS packets are a lot smaller then the data frame it makes sense to use them because if there is a collision, only the bandwidth for the smaller frames will be wasted rather then having the data frame take up a lot of time on the channel although it can not be sent successfully. When the data packet is small it may not make sense to bother with the overhead of the CTS and RTS packets.

Wireless networks typically contain a higher bit error rate than wired networks. However, they contain an automatic mechanism to resend lost packets. When the ACK frame is not received by the source station it can assume that the data frame was lost and that it should retransmit the data.

2.3 Software Architecture

The simulator software architecture is designed using object oriented programming with each component of the network stack being its own class in a fasion similar to J-Sim [JSIM04]. This allows for future modifications where different components can be swapped in and out for different types of networks. The simulator is run as a single thread and uses its own clock for timing. Although the simulation is coded is Visual C++ for Windows the only Windows specific elements are the user interface and the single thread that is used to run the simulation. Visual C++ is chosen over other languages such as Java because network simulators are processor intensive and c++ in network simulations is found to be ten times faster then Java [Nico02].

To be realistic, the network simulation is run on a clock. The physical layer clock (e.g at 11 mbits) is the clock used to time the network. This does not mean that the simulator has a loop that runs 11 million times for each second of execution. To speed things up, the simulation uses an event driven architecture. This means that rather than simulating every fraction of a second the main loop uses event based execution. An example of this is when transmitting a packet. Once it is verified that the device can transmit successfully, the simulation clock is simply advanced by an amount equal to the time required for the packet transmission. When there is nothing to transmit, the simulator advances to the next instance where it releases control to the processes.

The simulator has some advanced features. The simulator is designed as a virtual network meaning it allows for complete client server applications that can be run as separate processes and actually send data through the simulated network. The simulator ensures data integrity so it is possible to send complete files through the virtual network and the sent files will be the same as the original files. The simulator also allows for multiple channels to be used. Multiple channels can be used to simulate future versions of 802.11, using more than one channel at the same time to transmit packets.

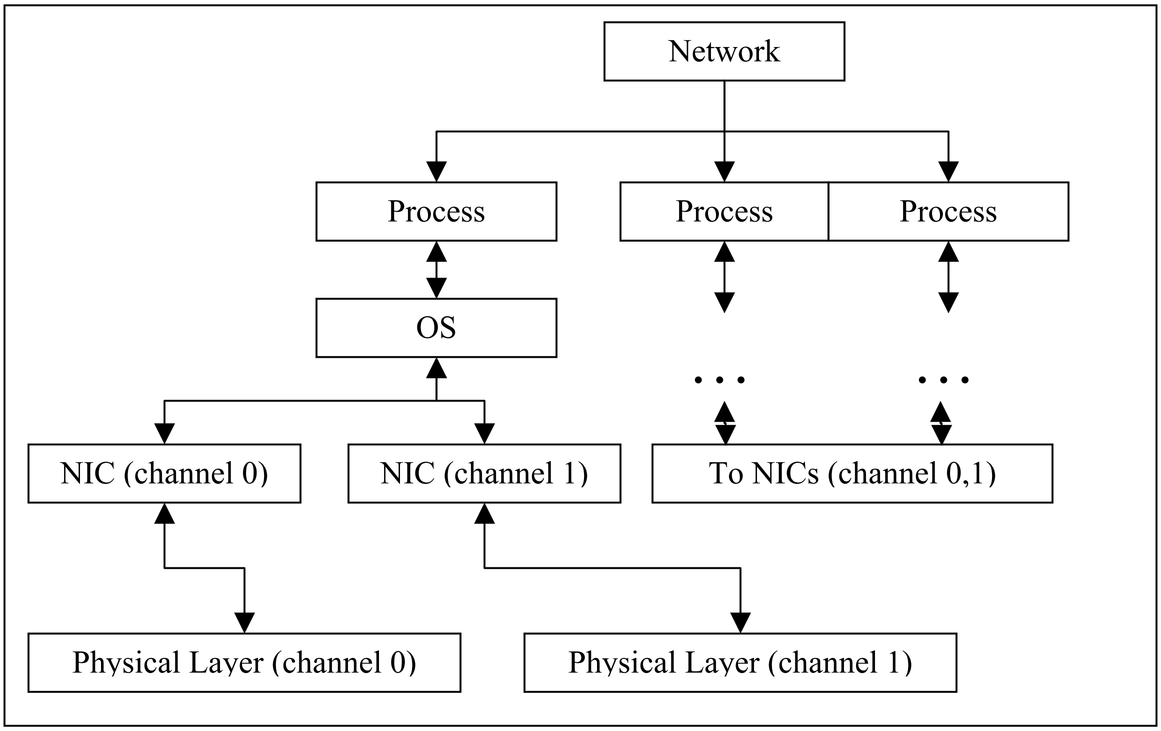


Figure 2-3 Architecture showing multiple channels

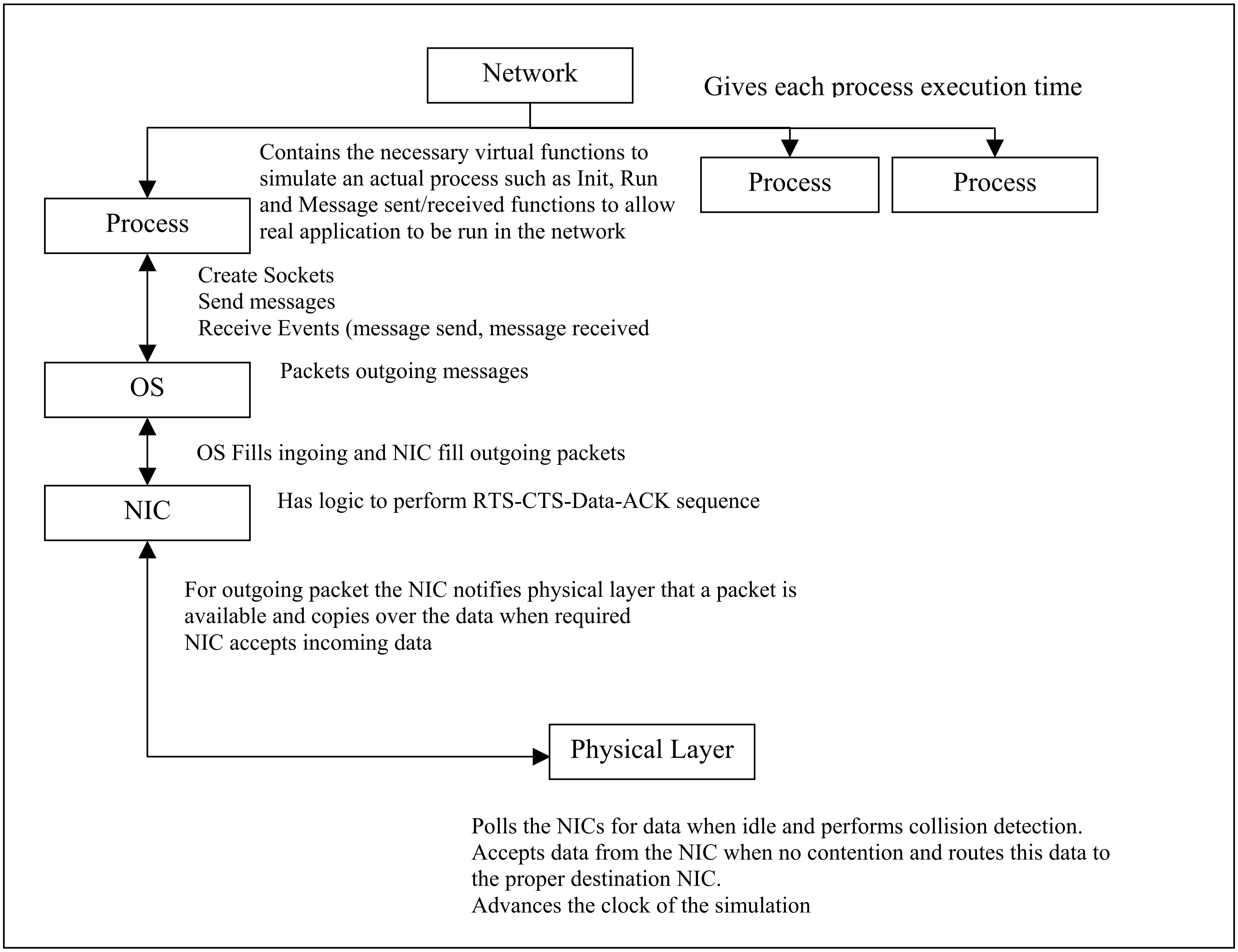

Network
Gives each process execution time
Contains the necessary virtual functions to simulate an actual process such as Init, Run and Message sent/received functions to allow real application to be run in the network
Process
Process
Process
Create Sockets
Send messages
Receive Events (message send, message received
OS
Packets outgoing messages
OS Fills ingoing and NIC fill outgoing packets
NIC
Has logic to perform RTS-CTS-Data-ACK sequence
For outgoing packet the NIC notifies physical layer that a packet is available and copies over the data when required
NIC accepts incoming data
Physical Layer
Polls the NICs for data when idle and performs collision detection.
Accepts data from the NIC when no contention and routes this data to the proper destination NIC.
Advances the clock of the simulation

Figure 2-4 Architecture with one channel but detailed interactions

The organization of the architecture is shown in Figure 2-3 and Figure 2-4. Figure 2-3 shows an example of the architecture with multiple processes and 2 channels. The number of channels and processes can be changed to any quantity. Figure 2-4 is a more detailed Figure that shows the interaction between the classes of the simulation, but does not show the multiple channels.

Figure 2-4 shows the interactions between each layer in the simulation. Each layer is its own class. The network class contains processes, each of which represent a stand alone computer. The simulation runs a loop in a single thread by alternatively giving the physical layer component run time and then giving processes run time. The processes generate packets that are first sent to the Operating System (OS) component. The OS component adds a layer of headers and then selects (if more then one channel) a Network Interface Card (NIC) component and sends that packet to the NIC. Then another layer of headers are added. When the physical layers runs, it checks each NIC to see if it has data ready. If it does and if there are no collisions, it copies the packet from the source NIC to the destination NIC. The packet propagates up the NIC to the OS, which then triggers an event in the process telling the process that the packet is ready. The process then can read the packet and send a reply if necessary.

The overall architecture looks a lot like how a computer on a network is organized. This is not an accident. There are two reasons for this, software engineering and realism. One of the main goals of software engineering is maintainability of code. When designing large programs it is best to separate the code into various classes that resemble different logical functions of the overall program. Each class should be a black box with well defined input and output. An internal change in one of the classes should not affect the other. If it would, it would create a debugging nightmare, in which one change might affect another and so on. The best example of the merit of using this approach in my system is the multiple channel addition. The simulation was not intended to support multiple channels when it was first conceived. However, because the simulator is modular, it is very easy to add the multiple channels by simply adding as many physical layers as desired, and having each OS support multiple NICs, one for each physical layer.

The other reason for this software architecture is realism. It is desired that this simulation be as close to the real thing as possible. This means that having actual traffic sennt over the simulation is desired. To do this it must be possible to write programs that can run on the simulation. This requires that there must be an Application Program Interface (API) and that data integrity must be maintained in the simulation. Using this architecture makes it easier to implement the API.

The realism is further enhanced with the simple model, complex behavior ideology. Each class in the simulation is not too complex and does not contain too many functions. However, each class closely resembles the actual functionality of the device it is emulating. This allows for natural complex behavior to occur without having to hardcode it. An example of this is what happens when a CTS packet is not received. Once the DIFS is passed, after the CTS is supposed to arrive, the channel becomes open again to retry. Rather then having to program this explicitly in one monolithic (complex) sequence it occurs naturally thought a number of simple interactions.

The following sections provide a more detailed look into how the simulation works internally. The section is split into three different areas. The first is how packets are sent over the network using the CSMA-CA, RTS-CTS-Data-ACK procedure. The physical layer acts to send the packets without knowledge of their format, while the NIC provides the RTS-CTS-Data-ACK state machine and follows the necessary procedures to send packets. The NIC class also has the option to become an access point. In the second area the OS and Processes class that are responsible for creating the API, that allow for processes to generate realistic traffic are discussed. The third area covered is the Network class, which is responsible for the overall execution of the experiments. It allows for incremental experimentation to be easily done, such as increasing the traffic over a number of runs and storing the statistics of each run to see the effects.

2.3.1 Physical Layer

The Physical layer is implemented in a base class that provides basic functionality, plus an child class, that provides the specific functionality of CSMA-CA. The physical layer performs the following functionality:

1) The physical layer provides the main clock of the simulation. It is responsible for all of the timing.
2) Generates bit errors according to a bit error rate function
3) Set up the event that gathers statistics
4) Copy packets from one NIC to another.

The base class provides functionality that is common to any physical layer implementation. To query if the device has a packet to send, the physical layer needs to know every device that is connected to it. It contains the functions *RegisterAP(NIC *pNIC)* and *RegisterNIC(NIC *pNIC)* depending if the NIC is conFigured as an access point or not. These functions allow each NIC class to register with the physical layer. The function *SetBER(int ber)* is used to set the bit error rate of the channel, while *CheckBitErrors()* is used to decide if the packet is to have a bit error. The function uses random numbers generated from the standard generator and the packet size to determine if the packet is to have a bit error. There are more complex error models possible such as, burst errors and error models for moving stations [HBEI01] but a simple function is sufficient for this thesis. By encapsulating the bit error rate functionality into two functions it is relatively easy to add a more complex bit error function such as one that is variable and changes over time. The class also contains the helper function, *GenerateStatistics(),* which is used to gather the statistics for an instance in time. It also contains the helper function *IncrementTime(int uSeconds),* which is used to advance the clock a certain amount of microseconds.

In order for the data to be actually sent, the *SendData()* function is used to copy the data from one NIC to another. This function makes use of the fact that the packets are framed with a header similar to the header used for the physical layer in 802.11. The header contains the following structure.

```
#define RTSFrame 01
#define CTSFrame 02
#define ACKFrame 03
#define DataFrame 04
#define FRAMECONTROL int
enum LANTYPE {BroadcastType, ConnectionType};
struct LANINFO
{
        FRAMECONTROL FrameControl;
        MACTYPE SourceMac;
        MACTYPE DestMac;
        MACTYPE TrueDestMac; // the true destination
        LANTYPE LanType;
};
```

The header contains the frame type, and 3 MAC addresses. There are three because of the access point functionality. There are two options when sending a packet, to either send it ad-hoc or to send it through the access point. When sending it through the access point the frame contains two destinations, the true destination of the packet plus the access point MAC address, to which the packet is sent to first.

The *SendData()* function is as follows:

```
void PhysicalLayer::SendData()
{
        CheckPacketType()
        for (int ix = 0; ix < m_NumNIC; ix++) {
            MACTYPE MacAddress = m_aNIC[ix]->GetMacAddress();
            if (MacAddress !=SourceMac) {
                    if (LanInfo.LanType == BroadcastType)
                         m_aNIC[ix]->SendData(m_CurrPacket,m_nCurrPacketSize);
                    else if (LanInfo.LanType == ConnectionType)
                         if (MacAddress ==DestMac)
                              m_aNIC[ix]->SendData(m_CurrPacket,m_nCurrPacketSize);
               }
          }
     }
```

The function first finds out the type of the packet. It then checks every NIC to see if it is the destination. If the packet is a broadcast type every NIC gets the packet. Otherwise only the destination NIC gets the packet. This is slightly different from how it works in reality, as every NIC on a shared channel gets every packet and those that the packet is not destined for simply discard it. However for simulation it is quicker if only the NIC that is supposed to get the packet does as it saves a lot of processing.

The child class provides most of the physical layers functionality. This is because there is support for changing the physical layer from CSMA-CA to CSMA-CD if required. The inherited class has only two functions: *Run()* and *GetNextPacket()*. Understanding these functions is integral to understanding how the simulation works. The simulation is run in a single thread. In the thread, the physical layer's *Run()* function is called while the thread loops repeatedly. The physical layer's *Run()* function runs for a certain quantity of time before returning, thus allowing for the processes to run. This quantity of time should be set to an intermediate value based on the speed of the physical layer. A larger time means that the simulation will be faster but less accurate. A smaller time means that the simulation will be slower but more accurate and approach the accuracy of a simulation that is not event based. In this case, the quantity of time is set to 1 millisecond as that is around the scheduling period of most operating systems.

The accuracy degrades when using multiple channels. When using multiple channels, a message may be sent over one channel and its reply in the other. However, because the channels are run for one quanta (a discrete quantity of time that a process or function is allocated), one after the other, the reply will not be received until the second channel gets its quanta. This means that the send-reply process on multiple channels is slowed down to one message per quanta rather then the possibility of it occurring several times within one quanta. Other than reducing the quanta size, which would slow everything down, the API programming could be made aware of this and send replies on the same channel as the original message. A far more complicated system of altering quanta size based on traffic could be used to alleviate this situation but it is not in the scope of this project.

```
void CSMACA::Run()
{
   // while we still can do operations before giving up control
    while (m_CurrBitTick < m_CurrBitQuanta)
    {
        if (m_bPacketToSend)
        {
            if ((m_nCurrPacketSize - m_nCurrPacket) > BytesLeft)
            {
                // only send some of it
                m_nCurrPacket += BytesLeft;
            }
            else
            {
                m_bPacketToSend = false;
                m_CurrBitTick += (m_nCurrPacketSize - m_nCurrPacket)*8;
                SendData(); // actual send the data to the other devices
            }

        }
        else
        {
            if (!GetNextPacket())
                m_CurrBitTick = m_CurrBitQuanta;
        }


    if (m_CGenStatsTime == m_GenStatsTime)
    {
        // generate statistics
        m_StatsNumTimes++;
        GenerateStatistics();
        m_CGenStatsTime = 0;
    }

}
```

The *Run()* function loops until its quantity of time is up. When it has idle time the function checks to see if there are any packets to send. If none of the NICs have any packets ready, the function can exit. If there is a packet ready the function will send it. If there is not enough time to send it, meaning that the time would exceed the quanta length, the function sends as much of it as possible by subtracting the size of the packet, and then in the next quanta the packet will be fully sent. That is the loop stops at exactly the time that its time is up, rather then running until the last message is fully sent.

The *GetNextPacket()* works with NIC functionality to see if any NICs have a packet to send. The code is long and complex so only a simplified algorithm is shown:

1) *IncrementTime(SIFS)* This is the function call to increment the time clock by a SIFS.
2) Check to see if any NICs have anyhing to send within the SIFS.
3) If still nothing to send *IncrementTime(PIFS-SIFS)*.
4) Check the access point to see if it has something to send
5) If still nothing to send *IncrementTime(DIFS-PIFS)*.
6) Query each NIC to see if it has something to send and at what time it wants to send it.
7) If there are none, there is nothing to send at this time, returns false.
8) If there is only one then that NIC is able to send the packet, return true.
9) If there are two or more NICs that want to send within the same slot time then there is a collision, inform them of the collision which activates their back off timers and keep on checking the NICs if any others want to send.

2.3.2 NIC Layer

The NIC class interacts with the physical layer and the OS classes. It acts as the intermediary. To interact with the OS it contains the function *EnterData(char *Buffer, int &nBufferLen)*, which allows the OS to copy data to the NIC in order to send it. To interact with the physical layer the NIC class contains several functions that the physical layer calls while it is running. *DataToSend()* is called by the physical layer to see if the NIC has a packet that it wants to send. The function returns the time at which the NIC wants to send it, from 0 to the maximum of the backoff timer. *DecrementBackoff()*, is used by the physical layer to decrement the backoff timer. The NIC also contains the utility function *GetBackoffFrames(int Resends)*. This function encapsulates all of the back off algorithm calculations. To change the back off algorithm only this function needs to be changed.

The major function in this class is *SendData(char *Buffer, int nBufferLen)*. This function is the one that the physical layer uses to notify that the NIC has received a packet. It is complex because it is responsible for the RTS-CTS-Data-ACK state machine, and access point functionality. The algorithm is as follows:

1) If a RTS frame is received, generate a CTS frame that is to be sent and prioritize it to be sent after a SIFS has occurred rather then a DIFS.
2) If a CTS frame is received, then queue up a data frame to be sent after a SIFS.
3) If an ACK frame is received, then call *ConfirmPacketSent(LANINFO LanInfo, IPINFO IPInfo)* which among other things tells the OS that the packet has been successfully sent.

4) If a data frame is received, check if it is addressed to the current NIC as the end point or as a relay to be repeated as the access point. The NIC has to be enabled as the access point in order to repeat the message.
5) If the NIC is the access point, change a few things in the message header and add it to the access point queue to be sent out.
6) If the NIC is the end point pass the packet up to the OS.

2.3.3 OS Layer

The OS layer is responsible for providing an API that is used when programming processes. It uses datagram sockets with a Transmission Control Protocol (TCP) like option for fragmenting. It supports fragmenting because a lot of the simulations send large packets and it is easier to write in the fragmenting code once in the OS level rather then having it written into every process.

The OS functionality looks a lot like a conventional OS. Examples are the following function calls, which are described later in the API section:

```
int OS::CreateSocket(int PortNumber, SOCKETTYPE SocketType, Process *pProcess)
int OS::Receive(int Socket, char *lpBuf, int nBufLen)
bool OS::SendTo(int Socket, int DestIP, char *lpBuf, int nBufLen)
void OS::SetSockOpt(int Socket, int Option)
```

The OS supports one additional feature. It supports multiple channels. In order to do this the OS creates multiple instances of the NIC class. Each NIC is assigned to a different physical channel. In order to choose which channel is selected the *PickChannel()* function is used. This is described in more detail in the experimentation section.

2.3.4 Process Layer

To program a process the *Process* class must be overridden. This class contains virtual functions that interact with other classes that provide the feel of operating in a real OS. A virtual function is one that is implemented in a base class such that when the base class is inherited, the function can be overridden by the inheriting classes. Virtual functions are necessary in message passing because the compiler needs to know that the virtual function exists somewhere. If the inheriting class does not override it, it calls the base class's version. This allows the base class to have access to potential useful functionality of the inheriting class. Examples of these functions are as follows:

```
virtual void OnMessageReceived(int SourceIP, int Port){};
virtual void OnMessageSent(int Port);
virtual void OnSaveStats() {}; // called once a second to generate stats
virtual void Run();
virtual void Init(); // called to do basic init (e.g setup the IP)
virtual void OnStart() {};
virtual void UserDrawStats(CDC *pDC, CPoint Offset, CPoint &Size);
```

OnMessageReceived and *OnMessageSent* have direct links to the physical layer. When a message is sent or received, an event (actually functions calling each other) is sent up the class hierarchy. At the end of this hierarchy the OS uses its process pointer to call the virtual functions which if implemented will perform user functionality. The other virtual functions are implemented similarly from different parts of the class hierarchy. The network layer calls many of the other process layer functions such as *Run* and *OnStart*.

2.3.5 Network Layer

The network layer is responsible for Domain Name Server (DNS) services, initialization, and running the simulation. The simulator supports simple DNS to make the application programming easier. The network layer has two functions to do this, *IPTYPE DNSFindAddress(char *Name)*, and *IPTYPE DNSGetIPAddress(char *&Name)*. *DNSGetIPAddress* adds a name to the DNS and returns the IP address. If the name is already there it returns a slightly modified name. *DNSFindAddress* returns the IP address given a name.

The simulator is a Windows application that uses the MFC (Microsoft Foundation Classes), including the way MFC handles threading. The simulation code consists of a single thread. The function *SimulatorThread* is shown below:

```
UINT CChildView::SimulatorThread(LPVOID lpInfo)
{
   THREADINFO *ThreadInfo = (THREADINFO *)lpInfo;
   DWORD SleepIn = 200; // sleeps every 200 ticks

   DWORD StartCount = GetTickCount();

   while (!ThreadInfo->EndSimThread)
   {
        for (int ix = 0; ix < 100; ix++)
        {
            ThreadInfo->ParentPtr->m_pNetwork->Run();
        }
        DWORD EndCount = GetTickCount();
        if (EndCount - StartCount >= SleepIn)
        {
                ThreadInfo->ParentPtr->Invalidate(false);
                ThreadInfo->ParentPtr->UpdateWindow();
                Sleep(25); // give up some time for user input
                StartCount = GetTickCount();
        }
        // check if the network still should run
        if (!ThreadInfo->ParentPtr->m_pNetwork->RunTestingCycle())
        {
                ThreadInfo->EndSimThread = true;
                ThreadInfo->ParentPtr->Invalidate(false);
        }

    }
   return 0;
}
```

The function is launched when the simulator is told to start, and ends either on user input or when the simulation has completed running its experiments. It has one interesting feature and that is that the thread runs continuously up to a time period defined by the variable *SleepIn*. *SleepIn* defines the number of milliseconds the simulation should run until sleeping. In Windows it is important for a thread to sleep every now and then in order that control can return to the user. If the thread does not sleep it can appear that the system is locked up.

The network layer class provides the *Run* function, which is called from the Windows thread. The *Run* function first gives each physical layer control. The physical layer will run until its quantity of time is up. At that point each process is given control. Note that both functions contained in *Run* are non blocking and return fairly quickly.

```
void Network::Run()
{
        // first thing to do is run the physical layers
        for (int ix = 0; ix < NUMCHANNELS; ix++)
        {
                if (pPhysicalLayer[ix] != NULL)
                        pPhysicalLayer[ix]->Run();
        }

        // next run all of the process (e.g)
        for (ix = 0; ix < m_NumProcesses; ix++)
        {
                m_aProcess[ix]->Run();
        }

}
```

Each process on the network is run at a fraction of the main clock, at intervals of approximately 1ms. The physical layer loop runs at a large multiple of the main clock before letting the processes run. This can potentially lead to a large problem. In a real network all the devices are operating on their own clocks, thus they are completely independent. With all of the process on the same clock, synchronization could occur that could invalidate the simulation. An example of this occurring is, if there are several servers on the network that transmit several packets a second. If the servers are all synchronized, far more collision occur if they were not synchronized. In order to avoid this situation a certain amount of randomness is introduced. When queuing up a packet in the NIC layer, the NIC can add a small amount of random time. This means that although all of the processes are running on the same clock they are all randomly out of phase with each other.

The network layer class supports two kinds of opeartion, continuous and multirun. In multirun, the network runs for several iterations, defined by the user controlled variable *NumMultiRuns*. For every new test interaction, the function *SetupNetwork* is called to setup the new network based on the current iteration. This type of operation is shown in the functions below. This main thread calls the function *RunTestingCycle* periodically. The function checks if the amount of time designated for the test is over. If it is, the function calls *RunNextTest*. This function checks to see if there are any more iterations of the test to complete. If there are, it resets the system and returns true. Otherwise if the experimentation is complete, it archives the statistics and returns false.

```
bool Network::RunNextTest()
{
	if (m_nSaveStats < m_NumMultiRuns)
	{
		m_nSaveStats++;
		g_NetworkStats.ArchiveStats();

		// now delete everything
		for (int ix = 0; ix < m_NumProcesses; ix++)
		{
			delete m_aProcess[ix];
		}
		m_NumProcesses = 0;

		Reset();
		for (ix = 0; ix < NUMCHANNELS; ix++)
		{
			if (pPhysicalLayer[ix] != NULL)
				pPhysicalLayer[ix]->Reset();
		}


		// now recreate everything
		SetupNetwork();
		return true;
	}
	else
	{
		g_NetworkStats.ArchiveStats(false);
		SaveNetworkStats();
		return false;
	}
}

bool Network::RunTestingCycle()
{
        if (TType == ContRun)
        {
                return true;
        }
        else if (TType == MultiRun)
        {
                if (SGetSeconds() > RunsPerRun)
                {
                        return RunNextTest();
                }

                return true;
        }
        return false;
}
```

2.3.6 Storing Statistics

Saving the simulation statistics to a file is very important. The simulator has a single class that performs most of the work. The class name is *NetworkStats*. This class stores all of the important physical layer statistics such as bit rate, collision data, and bandwidth taken up for different kinds of frames. It also has the option to add custom statistics that are gathered up and stored with the main ones.

The statistics gathering occurs at a programmable interval at which the *GenerateStatistics()* function is called, which is by default, set to the value of one second. During this one second, the statistics are entered through various functions in the physical layer. Every time a packet is sent, a collision occurs, or any other event happens, these are entered into the statistics class. At the end of one second several things happen. The current values are copied into another structure, which store the previous values for display. The GUI always displays the previous values rather then the current ones that are changing. Then, the current values are added to the averaging structure. This is the structure that is used for file output. At the end of an experiment, any statistics that are relevant to the current experiment are saved to a file that can then be loaded into Excel.

2.4 Application Programming Inteface

The goal of the simulator is to provide an API that is very similar to programming under a conventional operating system, such as Windows or Linux. The functions described below should be familiar as they emulate operating system calls.

2.4.1 Process Overrides

An application should override the following functions in the process class for the simulation to work properly:

void Init()

This function is called when the network is first initialized. It can be used to create sockets.

void OnStart()

This function is called when the network is initialized but right before anything is ran. It can be used to obtain IP addresses.

Void Run()

This function is called every 1 ms by the architecture to act as a thread that is running continuously.

*void UserDrawStats(CDC *pDC, CPoint Offset, CPoint &Size)*

This function is called by the network to allow the application to display data on the screen while running

2.4.2 Event Overrides

The following functions are ones that should be overridden in the process class if the application requires any event based message handling:

void OnMessageReceived(int SourceIP, int Port)

This function is called when a message has been received. The port number and source IP are passed through to distinguish which socket has generated the event.

void OnMessageSent(int Port)

This function is called when a socket message is completely sent via the network card.

void OnSaveStats()

This function is called, once a second, to allow the process to save any statistics into the archive. It provides a nice way to store simulation data so it can be automatically averaged and written out when a program is done executing.

2.4.3 Utility Function Calls

There is a class in the network that handles all of the statistics archiving. All of the base statistics, such as the physical layer efficiency and the network card's message counts are automatically stored and can be written out. However, to store any statistics that are generated by an application, the following functions are used *int GetCustomStats(Process *pProcess, char *pName)*

This function gets an index to be used for later archiving.

void AddCustomStats(int index, float value)

Given an index obtained with *GetCustomStats* this function adds a value into the archive.

2.4.4 DNS services

A network that dynamically allocates IP addresses requires a DNS service. The following functions implement the service:

*IPTYPE DNSGetIPAddress(char *&Name)*

This function returns an IPAddress given a name. If the name is not unique which it should be, this function adds a number at the end of the name because no two names in the network can be the same.

*IPTYPE DNSFindAddress(char *Name)*

Given a name this function returns the IP address.

2.4.5 Timers

Two functions are used to obtain the current simulation time.

ULONGLONG NGetTicks()

This function returns the current time in ms.

ULONGLONG NGetUTicks()

This function returns the current time in us.

2.4.6 Socket Functionality

These function calls are made to be similar to other operating system's socket calls. These work in tandem to the application override functions *OnMessageSend* and *OnMessageReceived*:

*int CreateSocket(int PortNumber, SOCKETTYPE SocketType, Process *pProcess = NULL);*

This function creates a socket and returns its ID.

void SetSockOpt(int Socket, int Option)

This function sets up the options for the socket.

*bool Send(int Socket, char *lpBuf, int nBufLen)*

This function broadcasts a message.

b*ool SendTo(int Socket, int DestIP, char *lpBuf, int nBufLen)*

This function sends a message to a specific address.

*int Receive(int Socket, char *lpBuf, int nBufLen)*

This function should be called in the *OnMessageReceived* event function. It returns the data that is received.

2.4.7 Network Setup Functions

The network class only requires two functions to be overridden:

void SetupNetwork()

This function is used to conFigure a network. A typical network would create its processes and set their options.

void SaveNetworkStats()

This function is called at the end of a testing run. It saves the statistics of the network. It should be overridden so the user can specify exactly which statistics to save.

The network has the following options that should be set that determine the nature of the testing run (TType = MultiRun or ContRun). The network can be setup to run continuously or be setup to run a certain amount of times then stop running and output its statistics

RunsPerRun

This variable determines how many seconds each run in multirun runs for.

m_NumMultiRuns

This variable determines how many runs it does in multi run. For example run 5 times for 100 seconds each.

m_nSaveStats

This variable is generated internally and indexes which run is occurring. It is used to index the archiving of the statistics but can also be used to adjust the setup in the *SetupNetwork* function. An example of the use of this variable would be to increase traffic as *m_nSaveStats* is increasing so networking performance testing can be batched together.

Chapter 3 Wireless Simulation Experimentation

Before any experimentation can begin the simulator has to demonstrate that it works correctly. To verify this two experiments are run. The first is to verify that the theoretical maximum throughput of the simulation is similar to the real thing, and the second is to verify that data integrity is maintained.

After the simulator is verified to be working correctly, several experiments are performed. The first is to examine the transmission properties of 802.11, such as the effect of having to relay packets through an access point. The second is to evaluate what happens in a live streaming situation when all of the bandwidth is used up. The third experiment is to evaluate some theoretical situations such as to increase the number of channels used and see what happens.

3.1 Verification of the simulator

3.1.1 Bit Rate Test

In order to verify that the simulation is working properly several tests need to be run. One of them is to verify that the bit rate is accurate. In order to do this a test network is created where there is one server and one access point. The server broadcasts data to the access point, which is then repeated. For each iteration of the test, the server increases its traffic. The traffic starts at 40000 bytes a second and increases linearly for each iteration (40000*iteration number).

To demonstrate how easy the API makes programming the simulation, some of the code is displayed. In order to select which network to use, the *TestNetwork* is selected in the constructer of the main Windows class. This “switches” the *TestNetwork* class on, allowing the class to control the simulation. To quickly change which network is being simulated only the network class has to be changed.

```
CChildView::CChildView()
{
        m_pNetwork = new TestNetwork;
        m_pNetwork->Init();
}
```

The *TestNetwork* overrides the *Network* class that does most of the work. The *TestNetwork* class only has two functions, *SetupNetwork()* and *SaveStats()*. *SaveStats* just takes values of the *NetworkStats* class, that are automatically gathered, and saves them to file. *SetupNetwork* is fairly simple. It loads up three processes and initializes them. Notice that the data rate is variable depending on the iteration.

```
void TestNetwork::SetupNetwork()
{

        TType = MultiRun;
        m_NumMultiRuns = 15;

        m_aProcess[m_NumProcesses] = new ServerProcess;
        m_aProcess[m_NumProcesses]->Init();
        ServerProcess *tSProcess = (ServerProcess *)m_aProcess[m_NumProcesses];
        tSProcess->SetDataRate((m_nSaveStats+1)*40000);
        m_NumProcesses++;

        m_aProcess[m_NumProcesses] = new APProcess;
        m_aProcess[m_NumProcesses]->Init();
        m_NumProcesses++;

        m_aProcess[m_NumProcesses] = new ClientProcess;
        m_aProcess[m_NumProcesses]->Init();
        m_NumProcesses++;
}
```

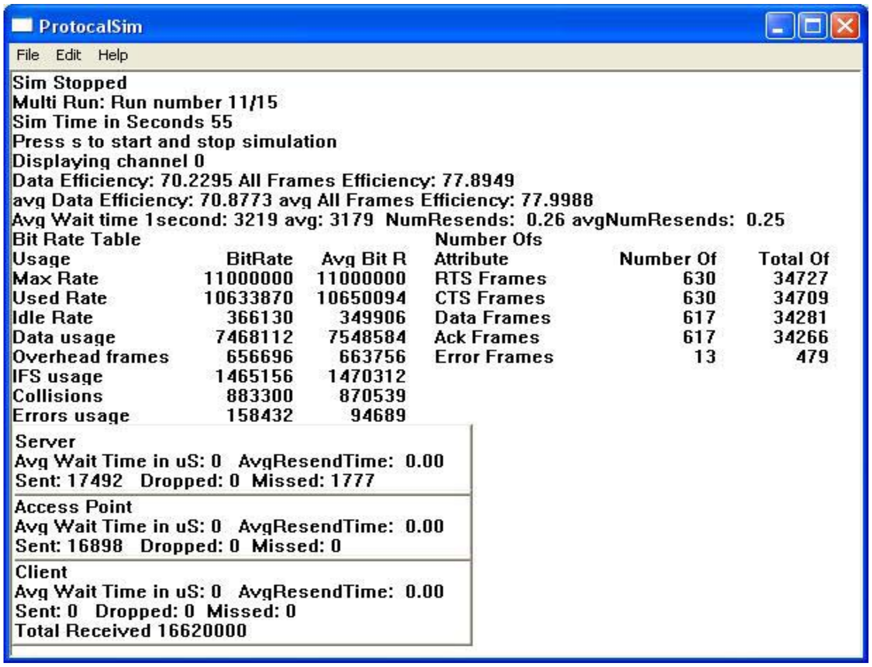
The *APProcess* is an empty class that just initializes a NIC to be an access point so no further explanation is required on that class. *ServerProcess* contains all of the key functionality which only requires three functions. The *Init* function creates a socket that is used for broadcasting. The *SetDataRate(int BRData)* allows the network to change the bit rate. The bit rate is changed by specifying the in-between time of two transmissions. The *Run* function sends out packets at this interval. The interval is selected partially randomly to make it more realistic. Notice that packet sizes of 10000 bytes are used and these are split up by the OS. This is done to verify that the fragmentation functionality works.

```
void ServerProcess::Run()
{
        m_CurrTime = NGetTicks();
        if (m_CurrTime >= m_SendTime)
        {
                // send something out
                int Rand = rand();
                int ExtraTime = (m_IntervalTime*Rand)/RAND_MAX;
                m_SendTime = m_CurrTime + m_IntervalTime/2 + ExtraTime;
                if (m_SendSocket != -1)
                        m_pOS->Send(m_SendSocket,m_SendBuffer,10000);
        }

}

void ServerProcess::Init()
{

        strcpy(m_ProcessName,"Server");
        // init create the socket
        Process::Init();
        m_SendSocket = m_pOS->CreateSocket(0,SockDataGram);
        m_pOS->SetSockOpt(m_SendSocket,BROADCAST);

}

void ServerProcess::SetDataRate(int BRData)
{
        // set it up to match the bit rate approx
        int MPerSecond = BRData/10000;
        if (MPerSecond <= 0)
                MPerSecond = 1;
        m_IntervalTime = 1000/MPerSecond;
}
```

The *Client* process does not contain a *Run* function because all it does is receives data. It contains two functions of interest, *OnMessageReceived, and UserDrawStats*. These two functions are virtual function overrides. *OnMessageReceived,* receives a message sent by the server through the access point and adds the length of the message to a counter. *UserDrawStats,* displays the total number of bytes sent by the server. Notice in Figure 3-1 that the client has an extra item in its display table that is generated by *UserDrawStats*. All of the formatting is done automatically.

```
void ClientProcess::OnMessageReceived(int SourceIP, int Port)
{
        int length = 20000;
        char blah[20000];

        int length1 = m_pOS->Receive(m_ListenSocket,blah,length);
        m_TotalReceived += length1;
}

void ClientProcess::UserDrawStats(CDC *pDC, CPoint Offset, CPoint &Size)
{
        char tempo[200];
        sprintf(tempo,"Total Received %d",m_TotalReceived);
        pDC->TextOut(Offset.x+2,Size.y,tempo);
        Size.y += 15;

}
```

Figure 3-1, shows the visualization of the test. Most of the important data is displayed in the GUI.



```
Sim Stopped
Multi Run: Run number 11/15
Sim Time in Seconds 55
Press s to start and stop simulation
Displaying channel 0
Data Efficiency: 70.2295 All Frames Efficiency: 77.8949
avg Data Efficiency: 70.8773 avg All Frames Efficiency: 77.9988
Avg Wait time 1second: 3219 avg: 3179  NumResends:  0.26 avgNumResends:  0.25
Bit Rate Table                          Number Ofs
Usage                BitRate  Avg Bit R  Attribute        Number Of    Total Of
Max Rate            11000000  11000000   RTS Frames             630       34727
Used Rate           10633870  10650094   CTS Frames             630       34709
Idle Rate             366130    349906   Data Frames            617       34281
Data usage           7468112   7548584   Ack Frames             617       34266
Overhead frames       656696    663756   Error Frames            13         479
IFS usage            1465156   1470312
Collisions            883300    870539
Errors usage          158432     94689
 Server
 Avg Wait Time in uS: 0   AvgResendTime:  0.00
 Sent: 17492   Dropped: 0  Missed: 1777
 Access Point
 Avg Wait Time in uS: 0   AvgResendTime:  0.00
 Sent: 16898   Dropped: 0  Missed: 0
 Client
 Avg Wait Time in uS: 0   AvgResendTime:  0.00
 Sent: 0   Dropped: 0  Missed: 0
 Total Received 16620000
```

Figure 3-1 Verification test GUI

Figure 3-2 shows one of the results of the simulation. It shows the network data bit rate increasing as the number of test iterations goes up. Notice that the line is not entirely linear because the function that sends data in the server uses integer arithmetic, therefore it is not perfect. The maximum throughput bit rate in the graph is 7,579,227 bytes per second. This is in between the theoretical maximum given by 7.1 Mbps as the UDP rate [Athe03b] and the 8.0 Mbps experimental result in chapter 6. The maximum bit rate is a direct factor of the look up table numbers used for such things as minislot time, which might be implemented slightly differently by different hardware vendors. The 7.6 Mbps is overall consistent.

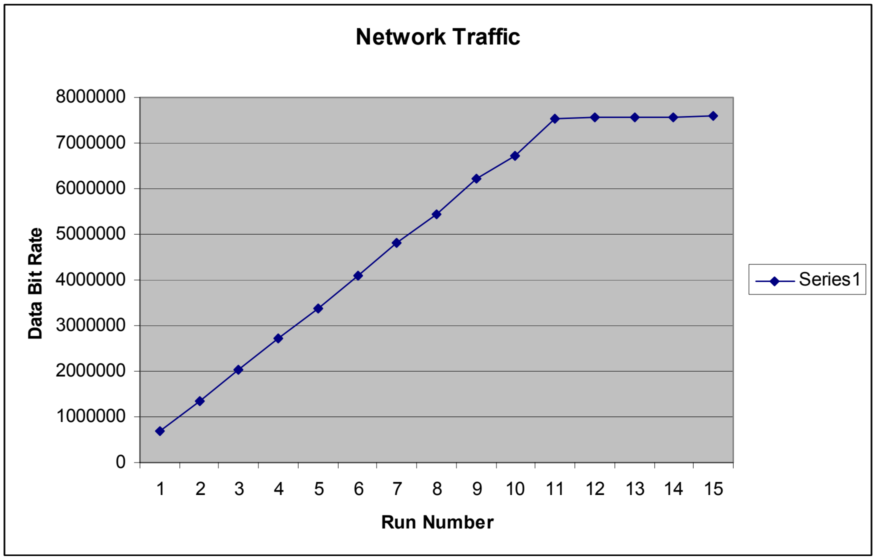


Figure 3-2 Verification results

3.1.2 Data Integrity Test

The simulation must maintain data integrity throughout. This means that if a file is sent through the virtual network, the copy sent on the network must be the same as the original. To verify this, a File Transfer Protocol (FTP) like setup is created. Where a client requests that a file be sent, and the server sends out a file. This example is more complex then the bit rate test, as it uses two sockets and handshaking. FTP generally requires a data port and a control port. The client sends a request to the server's control port. On receiving this message the server starts to send the file on the data port. When any fragment of the file is received by the client, the client sends out an acknowledgement. Upon receiving the acknowledgement, the server sends the next fragment. When the file download is completed, the server sends the client an end of file message on the control port and the client saves the file. To compare the two files, a file comparison program is used, which confirms that the sent file and original file are the same.

The *FTPNetwork* is quite simple as it contains a server, client and access point.

```
void FtpNetwork::SetupNetwork()
{

	RunsPerRun = 0; // sim done in 1 second set it as such
	TType = MultiRun;
	m_NumMultiRuns = 0;

	m_aProcess[m_NumProcesses] = new TServer1;
	m_aProcess[m_NumProcesses]->Init();
	m_NumProcesses++;

	m_aProcess[m_NumProcesses] = new TClient1;
	m_aProcess[m_NumProcesses]->Init();
	m_NumProcesses++;

	m_aProcess[m_NumProcesses] = new APProcess;
	m_aProcess[m_NumProcesses]->Init();
	m_NumProcesses++;

	Network::SetupNetwork();
}
```

The Client's *Run* function enters into an "if" statement that requests the file from the server only one time. After that, all the work is being done by the event handler, *OnMessageReceived*. If the packet is a data packet, it is appended to the file. If it is a control message, in this case the only control message, end of file, the file is written to the hard drive.

```
void TClient1::OnMessageReceived(int SourceIP, int Port)
{
    if (SourceIP == m_FTPServerIP && Port == DATAPORT)
    {
        // append to file
        m_CurrBufferLen += m_pOS->Receive(m_DataSocket,m_ReceiveBuffer+m_CurrBufferLen,200000);
        char tempo[200];
        sprintf(tempo,"Ack");
        m_pOS->SendTo(m_ControlSocket,m_FTPServerIP,tempo,strlen(tempo));
        // send out an ack
    }
    else if (SourceIP == m_FTPServerIP && Port == CONTROLPORT)
    {
        char tempo[200];
        m_pOS->Receive(m_ControlSocket,tempo,200);
        if (!strcmp(tempo,"EOF"))
        {
            // write out file
            FILE *tFile = fopen("ReceivedFile.jpg","w+b");
            fwrite(m_ReceiveBuffer,sizeof(char),m_CurrBufferLen,tFile);
            fclose(tFile);
        }
    }
}

void TClient1::Run()
{
    if (!m_bOnce)
    {
        m_bOnce = true;
        if (m_FTPServerIP != -1)
        {
            // send a request for file
            char tempo[100];
            sprintf(tempo,"Send File");
            m_CurrBufferLen = 0;
            m_pOS->SendTo(m_ControlSocket,m_FTPServerIP,tempo,strlen(tempo));
        }
    }
}
```

The server is also simple. In the *OnStart* function, which is called just before the network is run, the server loads up a file into memory, in this case a jpeg. All of the work in the server is done in the event handler, *OnMessageReceived*. Upon receiving the send file request the server begins to send the file. With every subsequent ACK, the server continues to send fragments of the file. When all the fragments are sent, the end of file message is sent to the client.

```
void TServer1::OnStart()
{
    // open up the file
    char tempo[200] = "c:\\ipclass\\protosim\\jan19.jpg";
  // char tempo[200] = "d:\\school\\ipclass\\protocalsim\\ReadMe.txt";
    FILE *tFile;
    tFile = fopen(tempo,"rb");
    m_TotalData = 0;
    if (tFile != NULL)
    {
        m_TotalData = fread(m_DataBuffer,sizeof(char),MAXBUFFER,tFile);
        ASSERT(m_TotalData < MAXBUFFER);
        fclose(tFile);
    }

}



void TServer1::OnMessageReceived(int SourceIP, int Port)
{
    char tempo[200];
    m_pOS->Receive(m_ControlSocket,tempo,200);
    if (Port == CONTROLPORT)
    {
        if (!strcmp(tempo,"Send File"))
        {
            m_ClientIP = SourceIP;
            m_DataSent = 0;
            int NumDataToSend;
            if (m_DataSent + m_DataSize > m_TotalData)
                NumDataToSend = m_TotalData - m_DataSent;
            else
                NumDataToSend = m_DataSize;
            m_pOS->SendTo(m_DataSocket,m_ClientIP,m_DataBuffer,NumDataToSend);
        }
        else if (!strcmp(tempo,"Ack"))
        {
            m_DataSent += m_DataSize;
            if (m_DataSent >= m_TotalData)
            {
                // just send an EOF message
                char tempo[200];
                sprintf(tempo,"EOF");
                m_pOS->SendTo(m_ControlSocket,m_ClientIP,tempo,strlen(tempo));
            }
            else
            {
                int NumDataToSend;
                if (m_DataSent + m_DataSize > m_TotalData)
                    NumDataToSend = m_TotalData - m_DataSent;
                else
                    NumDataToSend = m_DataSize;

                m_pOS->SendTo(m_DataSocket,m_ClientIP,m_DataBuffer + m_DataSent,NumDataToSend);
            }
        }
    }
}
```

The data integrity is verified by both doing a byte wise comparison of the files, and seeing if both files looks the same when loaded into a viewer. Both comparisons pass as the file is successfully sent over the network.

In addition to verifying the data integrity of the simulator, this example also shows how easy it is to create a realistic application running over the virtual network. Since the API of the simulator closely resembles an actual OS API, it would likely be able to use actual application code and run it over the simulator.

3.2 Simulator Experimentation

There are two reasons for using a wireless simulator. The first, is that it gives insight into how 802.11 actually works. Experiments can be set up to observe what happens internally in certain situations. The second, is to see what happens when the 802.11 standard is changed. 802.11 is designed as a general purpose standard. It may be possible to make better use of bandwidth with a standard designed specifically for media streaming. Since it is quite difficult to implement any changes to the standard in hardware, it is advantageous to evaluate any such changes using a simulator.

This section evaluates 802.11 using three experiments. The first, determines what differences occur when routing traffic through an access point compared to a peer to peer situations. The second, determines what happens when all the bandwidth is being used up. The third, performs experiments using multiple channels are used.

3.2.1 Effects of an Access Point

For media streaming with an access point there are two scenarios. The first, is that the media server is connected directly to an access point or the media is being streamed peer to peer through an access point. The setup with only one wireless connection is shown in Figure 3-3. The other setup is where the media server is streaming media through the access point as in Figure 3-4.

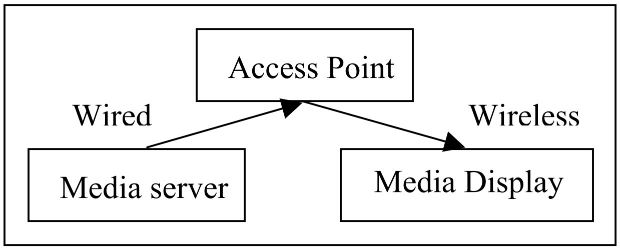


Figure 3-3 Test Setup 1

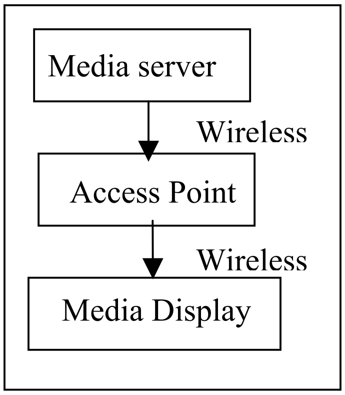


Figure 3-4 Test setup 2

Experiments are run to compare the various statistics in the two different setups. The results are shown in the following Figures, 3-5 to 3-8.

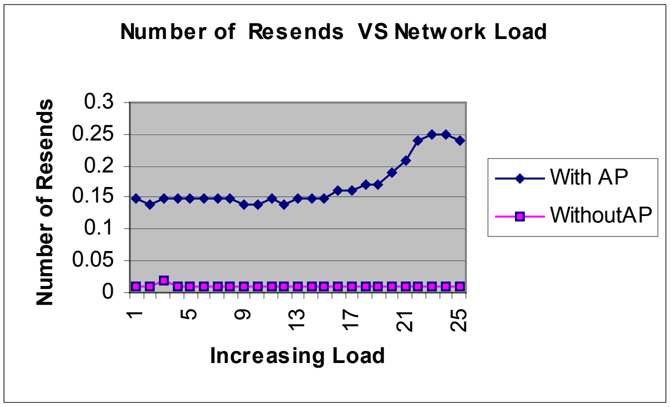


Figure 3-5 Number of resends VS network load

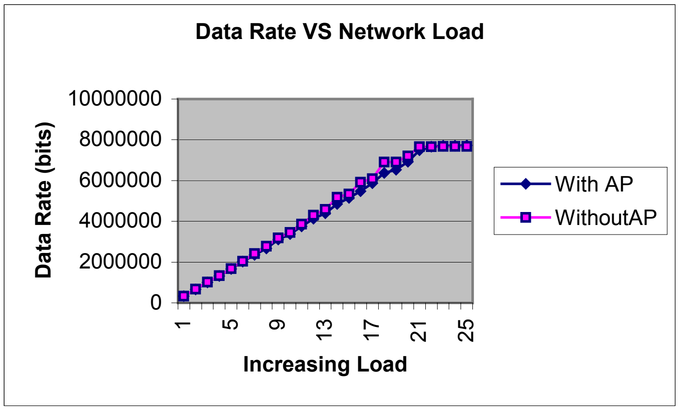


Figure 3-6 Data rate VS network load

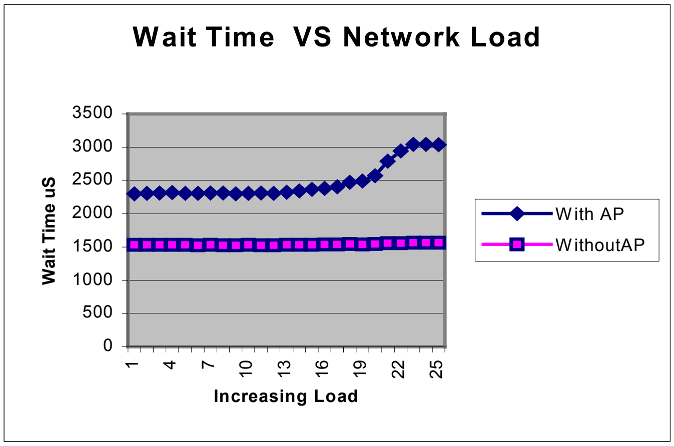


Figure 3-7 Wait time VS network load

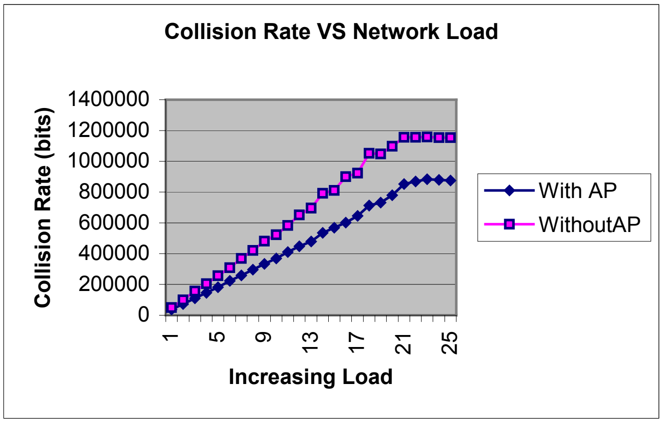


Figure 3-8 Collision rate VS network load

Figure 3-5 to 3-8 show the results. One of the setups has data routed through the access point and the other can be said to be peer-peer. There is only one server sending data. The data rate is increased to the point of network saturation. The x-axis in all graphs are test runs each of which increase in size of data sent. Each run increases the quantity of data linearly. Note that the access point routing experiment had only half the data sent to it to account for the access point resending all the data.

In terms of total data sent, the withoutAP is actually the same before saturation but because of a floating point error the withoutAP sends out more data frames then withAP as you can see with the drift in Figure 3-6. No packets are dropped or missed until saturation, so everything is getting through. After saturation withAP sends out slightly more data, 7.7 Mbps compared to 7.67Mbps. The reason for this is that the back off time for 802.11 is 2^{k+2}. That means for the first interval there is an equal probability of sending any 0-7 slots. However when two devices want to send, it is more likely that less time will get wasted. This is shown in Figure 3-8, that shows less time in terms of wasted bits waiting for something to send. This small amount though, is relatively insignificant.

The real difference is in wait time and number of resends shown in Figure 3-5. Here wait time is the time between when a packet is ready to send and when the data portion is sent. Both are higher in withAP then withoutAP. The wait time is calculated for only one trip, so overall latency is much higher with the access point. Also note that the sending process is generally synchronous. When the server has data to send, it sends out multiple packets. Upon receiving the first out of a group of packets the access point immediately requests use of the channel to retransmit the packet. Because of this, the access point and server are sending data at the same time even when the channel is not congested. So the extra wait time is due to the collisions with both the server and access point requesting the channel.

These results show that routing data through an access point halves the available bandwidth but does not contribute to any additional significant efficiency loss in terms of throughput. Having an access point does raise the wait time, thus increasing total latency to a value higher then twice the peer to peer latency.

3.2.2 Wireless Simulation of 802.11 Media Streaming

This section looks at what happens when video is streamed over 802.11. The experiments looks at the effects of bandwidth saturations with and without frame fragmentation.

The wireless video simulation contains a server and a client. The server code is fairly simple. It contains a *Run* function and an *OnMessageSent* function. The server waits for a certain interval and then sends out fake RTP packets which contains the timestamp and frame number but no real data.

```
void VideoServerProcess::Run()
{
    if (m_SendTime <= NGetTicks())
    {
        if (m_ReadyToSend)
        {
            char *tBuffer = m_SendBuffer;
            RTPINFO RTPInfo;
            RTPInfo.TimeStamp = NGetUTicks();
            RTPInfo.FrameNumber = m_FrameNumber;
            m_FrameNumber++;
            g_Protocals.CatRTPInfo(tBuffer,RTPInfo);
            m_pOS->Send(m_DataSocket,m_SendBuffer,m_FrameSize);
            m_ReadyToSend = false;

        }
        else
        {
            m_FrameNumber++;
            m_SkippedSending++;
        }
        m_SendTime += m_SendIncTime; // no jitter for now
    }
}

void VideoServerProcess::OnMessageSent(int Port)
{
        m_ReadyToSend = true;
}
```

The client is far more complex because it is responsible for collecting the statistics. The client is modeled as a live video player. It waits until a certain number of frames are available and then its starts to play the video. When the number of frames in the buffer falls off, it stops playing and waits for the buffer to fill up again. The client stores how many video frames are played and the run length. The run length is defined as how many frames in a row are playable or not playable. This is important as video compression can require the previous frame in order to play the current one.

```
void VideoClientProcess::OnMessageReceived(int SourceIP, int Port)
{
        if (Port == VIDEODATAPORT)
        {
                m_pOS->Receive(m_DataSocket,m_ReceiveBuffer,MAXMESSAGESIZE);
                m_VideoBuffer.AddData(m_ReceiveBuffer);
        }

}


void VideoClientProcess::Run()
{
        if (m_StreamingTime <= NGetTicks())
        {
                m_CurrBufferSize = m_VideoBuffer.GetNumFrames();
                m_BufferSizeC += m_CurrBufferSize;
                m_BufferSizeT++;

                ArchiveInternalStats();

                bool CanPlay = false;
                if (m_bBufferBigEnough)
                {
                        if (m_CurrBufferSize > m_MinHBuffer)
                                CanPlay = true;
                        else
                                m_bBufferBigEnough = false;
                }
                else
                {
                        if (m_CurrBufferSize > m_MaxHBuffer)
                        {
                                CanPlay = true;
                                m_bBufferBigEnough = true;
                        }
                }

                if (CanPlay)
        {
            char *tBuffer = m_ReceiveBuffer;
            bool ValidData = m_VideoBuffer.GetData(tBuffer);
            if (m_bPlayedLastFrame)
            {
                if (ValidData)
                {
                    m_TotalPlayedFrames++;
                    m_RunCounter++;
                    if (m_RunCounter >= 50)
                    {
                        m_TotalPlayedRuns++;
                        m_PlayedRunSizeC += m_RunCounter;
                        m_RunCounter = 1;
                    }
                }
                else
                {
                    m_TotalPlayedRuns++;
                    m_PlayedRunSizeC += m_RunCounter;
                    m_bPlayedLastFrame = false;
                    m_RunCounter = 1;
                }
            }
            else
            {
                if (ValidData)
                {
                    m_TotalPlayedFrames++;
                    m_bPlayedLastFrame = true;
                    m_TotalMissedRuns++;
                    m_MissedRunSizeC += m_RunCounter;
                    m_RunCounter = 1;
                }
                else
                {
                    m_RunCounter++;
                    if (m_RunCounter >= 50)
                    {
                        m_TotalMissedRuns++;
                        m_MissedRunSizeC += m_RunCounter;
                        m_RunCounter = 1;
                    }
                }
            }

        }
        else
            m_NumTimesSkipped++;

        m_StreamingTime += m_StreamingTimeInt; // no jitter for now
    }

}
```

The *OnMessageReceived* simply checks which port the data is coming from and then adds the data into a buffer. The *Run* function does all the work. It first checks to see if there is enough data in the buffer in order to play a frame. If it can play the frame it increments the number of frames played and alters the run length counter accordingly.

Note that the run length is arbitrarily capped at 50, which means that either the video is being continuously played or continuously not played.

In terms of the frame rate, the video frame size is identical. That is, having a frame interval of 45ms uses the least bandwidth and a frame interval of 30ms uses the most bandwidth. The values are chosen such that 45ms is right before bandwidth saturation and 30ms is an extreme example of bandwidth saturation. Note that for all the graphs, the x-axis is time. Each unit represents one video frame.

Generally speaking, when all of the bandwidth is used up, the buffers of the OS fill up. When this happens and a *SendTo* function is invoked, the OS cannot accept the data and returns an error message. At this point, the application can either wait for room to appear in the buffer or discard the packet. In this experiment, the application also had the option to fragment or not fragment the data. Since the simulator OS supports fragmentation, an option could be toggled that lets the OS discard the entire packet if it does not fit into the buffer rather then fragmenting it and sending only some of it.

The effects of fragmentation are shown by comparing the results in Figures 3-9 and 3-10. Here, the same streaming bit rate is used, however, in Figure 3-10 packets are only sent if the entire packet can be placed in the OS buffer. In Figure 3-10 it can be observed, that as the required bandwidth goes up, the number of frames that do not get through to the client increases. This increase appear to be linear. However when the packets are fragmented as shown in Figure 3-9, the increase can be described as catastrophic. That is, after a certain point very few or any whole video frames make it through.

The experiment gathers statistics on how the missing frames appear on the play buffer. This can be seen in Figures 3-11 and 3-12. These Figures show the play buffer and the run length. As can be seen in Figure 3-10, the 37ms frame interval has around a 60% play rate. This corresponds to the buffer being only around 60% full. This means that although by frame number there is a 30th frame in the buffer, there are several missing frames interspersed. Figure 3-12 shows the run length. On average there are 1.6 frames playing and then 1 frame not playing, then 1.6 framing played etc. This shows that the missing frames can be thought as holes periodically spaced in the play buffer.

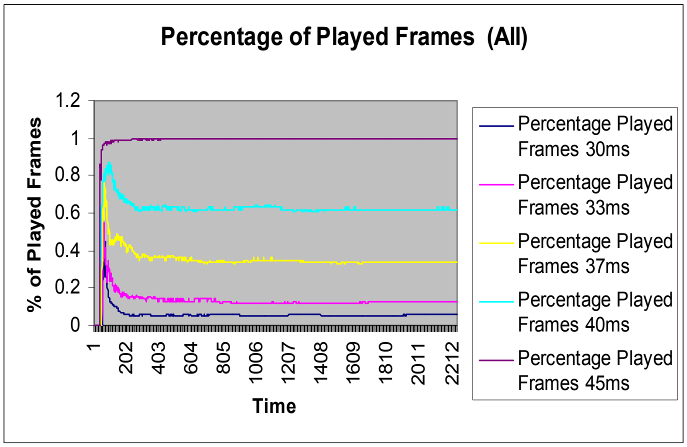


Figure 3-9 Percentage of played frames

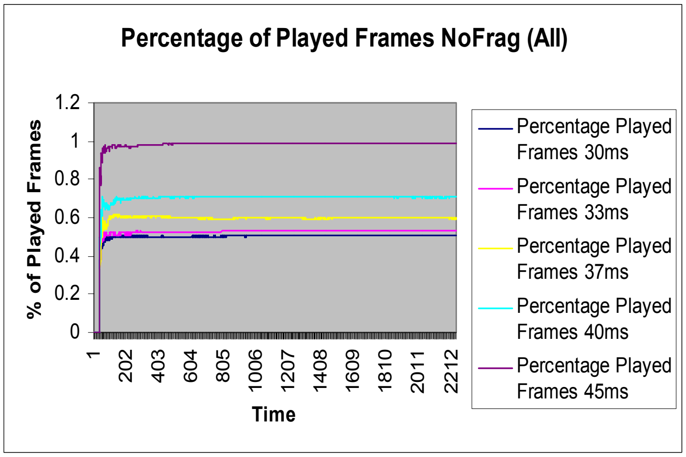


Figure 3-10 Percentage of played frames no fragmentation

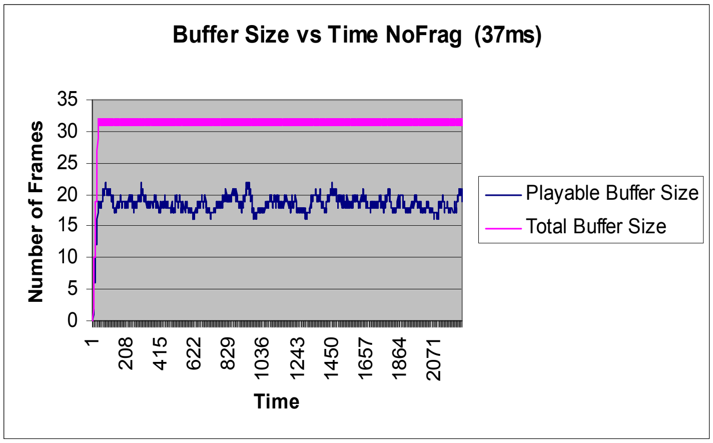


Figure 3-11 Buffer size

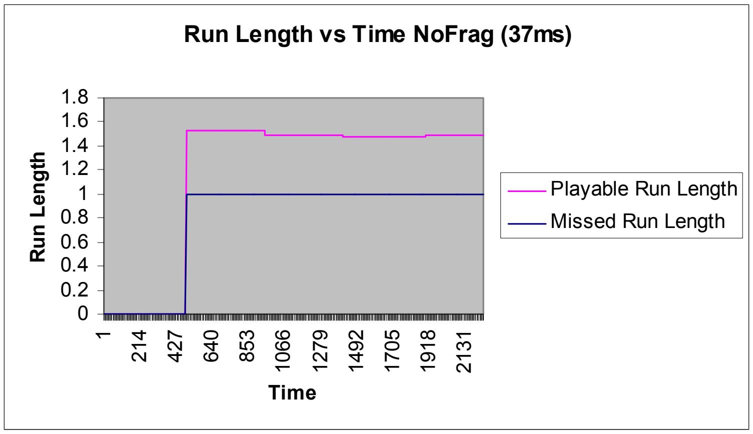


Figure 3-12 Run length

The result of this experiment shows several things. The first is the effects of fragmentation. Generally speaking, fragmentation should be avoided if possible. It also shows that when trying to send more data than the network can handle packets are dropped at the sender's end by the OS. These packets are dropped at a periodic rate, that is on average they are evenly interspersed. This leaves holes in the clients play buffer, which is not desired because compression techniques require a chain of frames to be able to play correctly.

3.2.3 Multiple Channel Experimentation

Theoretically, adding a second channel simply doubles the maximum bit rate. However, under certain conditions, having more then one channel can lead to inefficiencies if one channel is used more than another. The experiments examine some performance issues with having multiple channels with multiple servers and having multiple channels with different Bit Error Rates (BER).

The first experiment verifies that increasing the number of channels, all with the same BER, will increase the bit rate proportionately. The experiment uses four servers rather then one to increase the randomness of the traffic. The combined channel bit rate generally increases linearly as seen in Figure 3-13. However, during early experimentation something odd occurred. The algorithm that was first used to pick which NIC to send a packet was very simple. It simply picked the one with the smallest buffer size at the time. If they all are empty it defaulted to channel 0. Originally, this was considered as the best algorithm to use, as it should have equalized the load level over all of the channels. As seen by Figure 3-13, 14,15 using this selection algorithm did not equalize the channels in terms of data bit rate, wait time and number of resends. This algorithm favored the channels with lower numbers as the selection algorithms generally favors them.

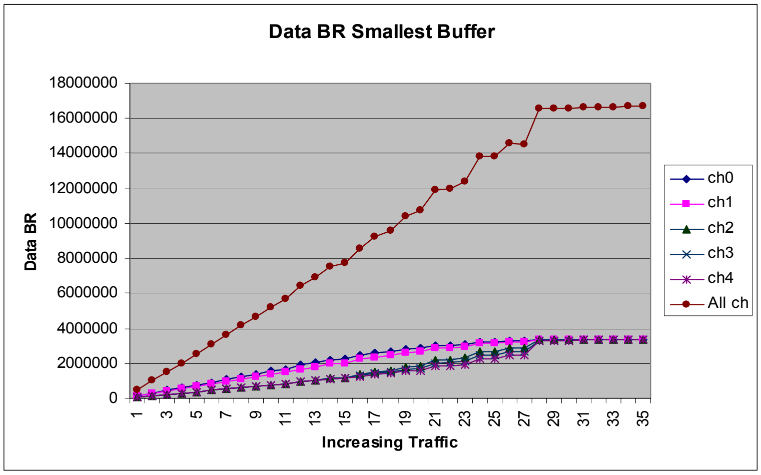


Figure 3-13 Data bit rate for five channels, smallest buffer selection

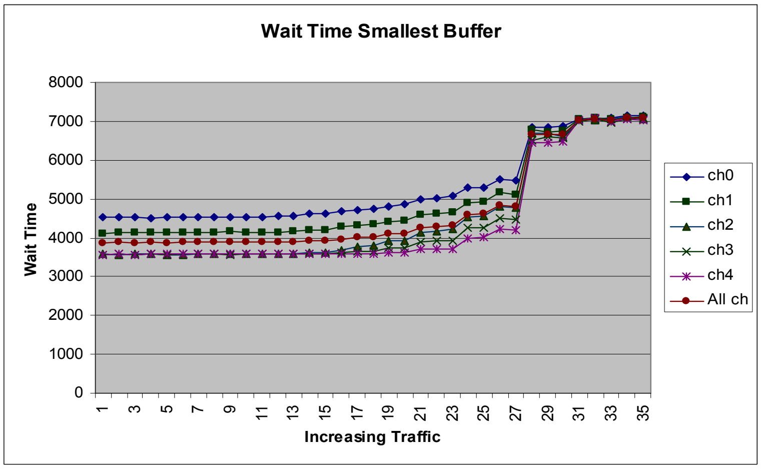


Figure 3-14 Wait time, smallest buffer selection

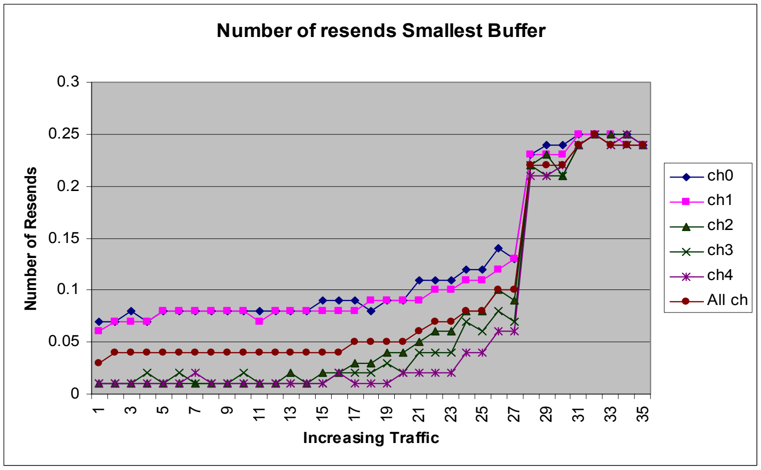


Figure 3-15 Number of resends, smallest buffer selection

The same experiment is re-run with a different channel selection algorithm. This algorithm instead selected the channels randomly, without any consideration of its load. The results are shown by Figure 3-16, 17, 18. This selection algorithm equalized all of the key statistics, which is desirable.

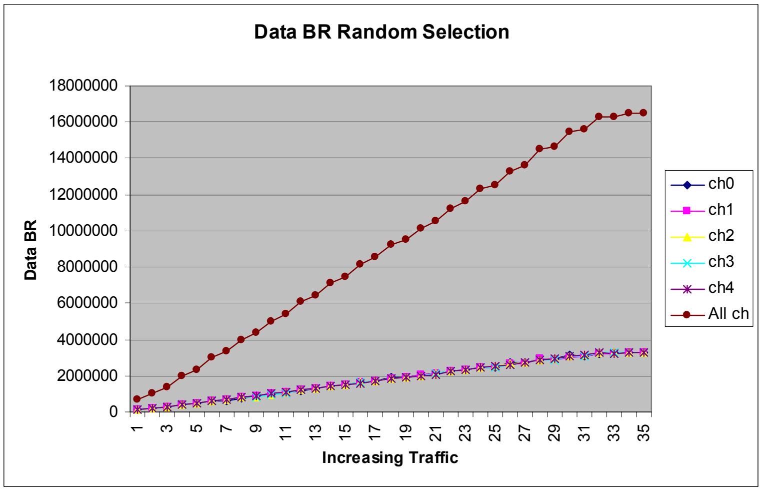


Figure 3-16 Data bit rate random selection

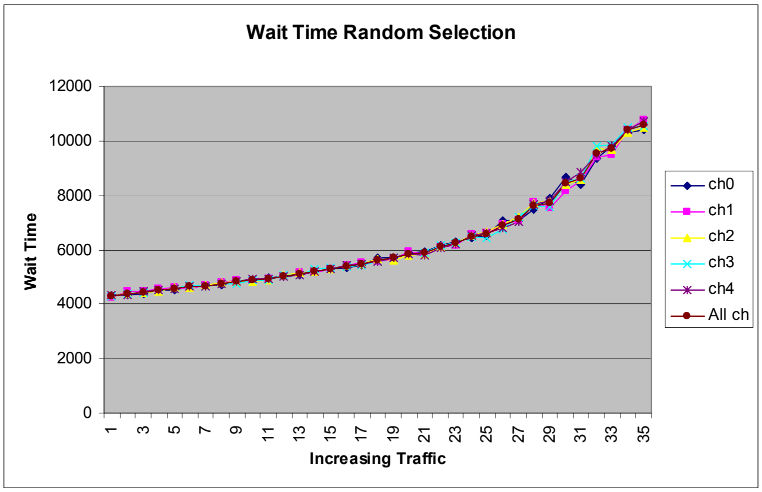


Figure 3-17 Wait time, random selection

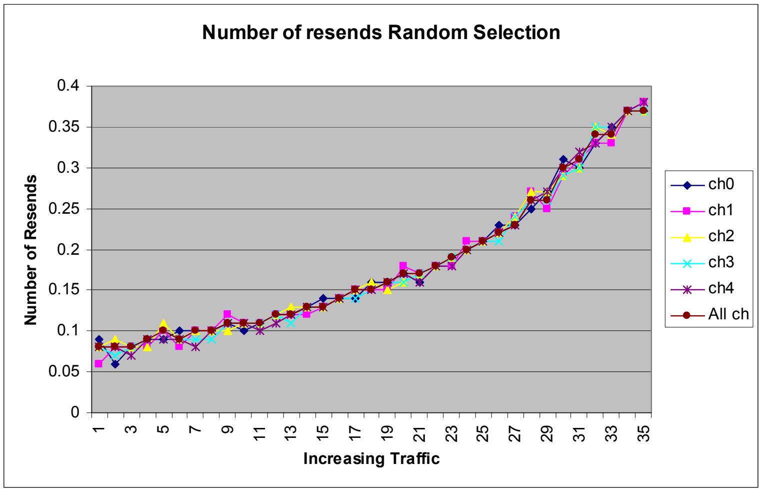


Figure 3-18 Number of resends, random selection

The next experiments are performed using two channels each with different BER, and applying the random selection algorithm. The results are shown in Figure 3-19,20,21. In Figure 3-19,20 the BER of channel 0 is set to 1 in 1,000,000 chance of any bit containing an error. The BER of channel 1is set to 1 in 20,000 chance of any bit containing an error. The results are dramatic, showing that the channel with the higher bit error rate has much higher wait time and number of resends. Figure 3-21 shows the results of a duel channel setup where one of the channels is fixed at 1 in a 1,000,000 bit error rate and the other have various bit error rates from 1 in 100,000 to 1 in 12,000. The results show that over higher network loads, the channel with the higher bit error rate gradually carries less of the load.

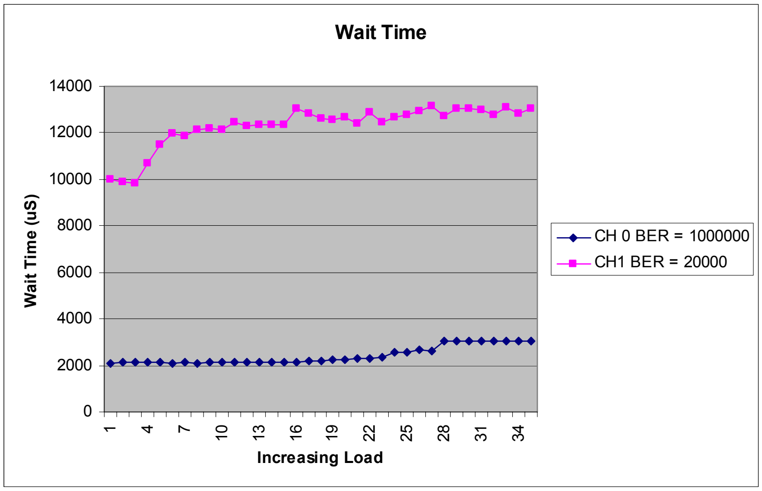


Figure 3-19 Wait time for different BER

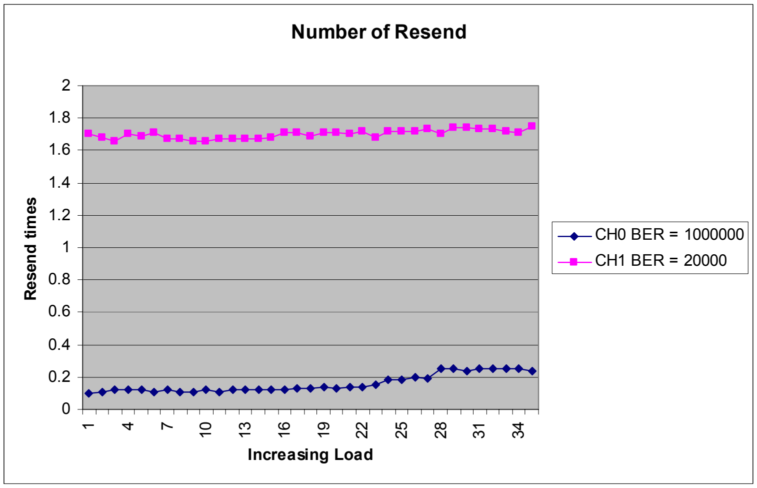


Figure 3-20 Number of Resends, different BER

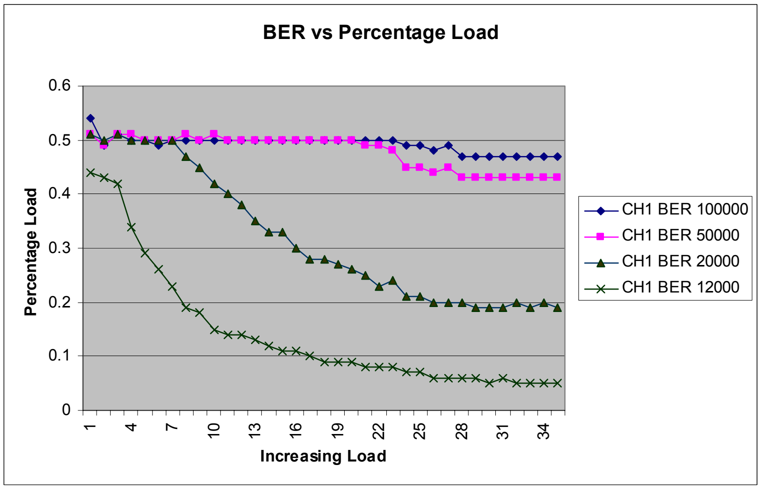


Figure 3-21 BER vs Percentage load

The last experiment is a comparison of the two channel selection algorithms when one channel has a higher bit error rate. These results are shown in Figure 3-22. It shows that using the smallest buffer is far better then choosing the random selection algorithm because too much data is placed in the channel with the high BER which never gets sent over. Note it is possible to use a hybrid algorithm that has the best of both algorithms. The results are meant to show the extremes when a poor algorithm is chosen.

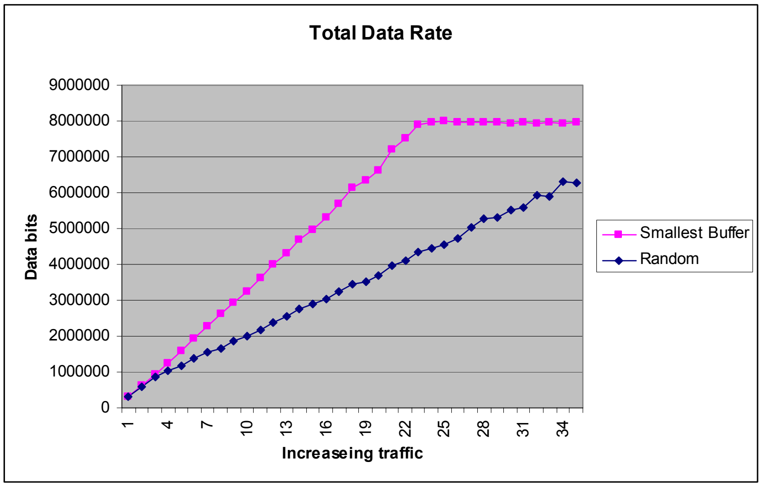


Figure 3-22 Total data rate of selection algorithms

3.2.4 Experimentation Conclusion

CSMA-CA (802.11) is quite a bit different from CDMA-CD (802.3) but the nature of its properties is similar. 802.11 generally, has lower throughput given a bit rate and higher latencies than 802.3. Its bit rate is also halved when routing communications through an access point. 802.11 is also susceptible to higher bit error rates that can severely impair performance. However in a well setup wireless network with a low bit error rate, other then a lower throughput and higher latencies, there are not any other properties of 802.11 that would cause problems for wireless streaming.

In terms of media streaming, it is important to make sure that when the packets are fragmented, that all of the data can be sent. This section also shows that doubling the number of channels can effectively double the bit rate. Some care though has to be given to the algorithm that selects the channel. It would be useful if there is some knowledge about the overall channel error rate that can be used in selecting a channel.

Chapter 4 DirectShow Filters

DirectShow is Microsoft's media streaming technology and is part of the larger DirectX suite. DirectShow contains all of the components for audio/video capture and playback in a multitude of formats. Windows Media Player is an application that is written using DirectShow. Most of the actual functionality in the media player is handled by DirectShow components.

4.1 Why Use DirectShow

A live media streaming application consists of many components, audio/video capture devices, encoders, decoders, renderers, the server and the data transport mechanism. Some of the components, such as the capture device drivers, are usually provided by the operating system or by the hardware manufactures. The audio/video renderers are also normally handled by the operating system. That leaves the encoder/decoder mechanism and the server/ data transport mechanism left to be implemented.

Although this project required the creation of the server/data transport mechanism, creating an encoder/decoder is beyond the scope of this thesis. Encoder technology has improved in recent years with the advent of MP3 audio, and MPEG-4 video. Creating a state of the art audio or video encoder from scratch is quite difficult. Instead pre-made audio/video encoder/decoders are commonly used. The solution chosen in this thesis is to use the Microsoft MPEG-4 video encoder/decoder [Micr05] and the LAME MP3 audio encoder [LAME05]. The MPEG-4 video encoder is provided free with Microsoft Windows XP. The LAME MP3 audio encoder has lower latency then the available Microsoft audio encoder Audio V8, and is freeware.

Both of the audio and video encoders are provided in DirectShow filter form. A DirectShow filter is a discrete component that has media functionality. By using the DirectShow API, it is possible to use the media capture, encoders/decoders and render components. This simplifies the overall development to only creating the server, the data transport mechanism, the client and the DirectShow components required to interoperate with the other existing DirectShow components.

4.2 Introduction to COM

For this project the Component Object Model (COM) [COM05] is used because DirectShow is based on COM. Each DirectShow object used, such as an encoder, a renderer or a capture device is a COM object. Although COM may be a hard to master, it is easy to understand the basics and program with it. The following section describes the basics of COM, in order that it can be understood when it is used in the next few sections.

COM base on a model of reusable components. The need for COM occurred as programs became more and more complex and it was realized that it is possible to reuse components rather then have to recreate the same code to different application. In the mid 90's Microsoft introduced COM as a way of using components in Windows. This can be seen in Microsoft software, such as the applications that make up the Office suite. COM is prevalent in applications such as Power Point and Word. Although Power Point and Word are different applications, they still contain similar components such as the spell checker, and the font editor. To load a COM object, the operating system, Windows, searches its record of all COM objects registered with it, and loads the proper COM objects if they exist.

COM is similar to the programming methodology of reusing code libraries. The difference is, that when using libraries, all of the code is recompiled into the new program. When using COM, the compiler only needs to know the interface to the object, and the object is created and accessed at run time. In this way, COM is similar to dynamic link libraries, however, as a standard it is formally defined. COM also has other features such as the ability for objects to be distributed, and to provide security access control.

A basic program contains data with a set of functions to manipulate the data. COM provided a black box approach to interfacing the data and the functions. COM objects contain interfaces, which defines how to interact with an object. The interface is simply a description of the callable functions that are implemented by the object. The interface is the only way to interact with the object as it is not possible to directly access data in an object, as it is possible to access public member variables from c++ class objects.

All COM objects implement the *IUnknown* interface. This interface provides the bare functionality needed for a COM object to exist. It contains three methods, *QueryInterface*, *AddRef*, and *Release*. *QueryInterface* allows the user to check for other interfaces of the object. *AddRef* and *Release* are required because of the way COM handles destruction of objects. It is possible for multiple clients to use the same object. When one client ends its use of the object, it cannot delete the object because then the object would be deleted for the other clients. Instead, the client calls *Release* which decrements an internal object reference count. When the object count reaches zero, it knows that no one is currently using the object and it can delete itself. The *AddRef* function increases the count of the object by one and is performed when the object is first referenced by a client.

In addition to the *IUnknown* interface, an object normally defines another interfaces that contain its functionality. The interfaces can either be a custom interface that is defined by the object or a standard interface that is used to represent common functionality of similar objects. An example of this is video encoders. There is a standard interface that all COM video encoders should implement that defines functionality relating to altering the encoder settings such as bit rate, frame rate, etc. However, if the encoder has any specific settings it can be implemented in its own custom interface.

In the following sections, the uses of COM are described in more detail. The video streaming project uses both pre-made COM objects such as video encoder/decoders, and also creates its own interfaces such as the audio renderer and the two DirectShow interface filters.

4.3 What is a DirectShow Filter

A DirectShow filter is a component that can be combined with other components to form a media stream. There are three main types of filters, source filters, transform filters and render filters. Source filters are filters that are the origin of media samples, such as files, capture cards, and internet streams. Transform filters are those that receive a media sample, transforms it by changing its format by compressing it or by editing the sample and sending it along to the next filter. Render filters are the last filter in a filter chain such as a video display filter or an audio render filter.

Filters are all designed using a common template. In the template, information such as the number of pins, the filter name, and the ID of the filter is stored. All the filters are based on COM, and depending on the filter type, each filter implements at least one or perhaps several COM interfaces that are specific to filters. Each filter contains at least one input/output pin. Each of these pins on the filter are also their own COM object, and handle the initial interfacing to other filters and the transportation of data.

The reason behind the filter architecture is so that the media streaming software can be made up of components. Media streaming software can be made up of several components from different vendors. This makes it easy for hardware manufactures to write software for their devices, as they only need to implement those components that are directly related to the hardware device. By conforming to the standard, their devices can interoperate with those of the other manufacturers. Third party software developers, for example encoding vendors, need only write the encoder and decoder and they can use any available components when writing their media player. For our purposes using DirectShow filters make it possible to use pre-existing components such as video capture filters, and encoders/decoders. Without the use of these pre-existing filters, it would be impossible (due to time limitations) to construct live video/audio streaming software.

Knowledge of programming filters is required for this project. The server and the client program cannot directly communicate with standard DirectShow filters. The server is required to receive input from the encoders, while the client program that receives media data through the LAN, sends data to the decoders. For the server and client to communicate with DirectShow filters, two custom filters, one on the server side and one on the client side is required. These filters act as the interface between the server and client applications that are programmed using Visual C++ and MFC, and DirectShow filters, which are programming using Visual C++ and COM.

4.4 Graph Edit

For useful processing to occur, a number of filters have to be connected. A COM filter graph (*IFilterGraph*) interface exists that can be used to connect filters. This can either be done at the coding level using the filter graph interface, or by using a graphical program called Graph Edit. Graph Edit is a graphical representation of the filter graph interface. It is useful in quickly creating and debugging filter graphs. The software has a nice interface in which any filter that is registered can be loaded and then connected to another filter. To connect one filter to another, one can simply left click, hold on an output pin, drag the mouse over an input pin, and release. If it is possible for the filters to connect together they will. If they cannot connect together, the Graph Edit program will use a mechanism called intelligent connect. Graph Edit will search for any combination of in intermediary filters that can connect the two filters together. If it cannot find one, it gives up and outputs an error message. If it can find some intermediate filters, it will automatically insert them into the graph and connect all the filters together.

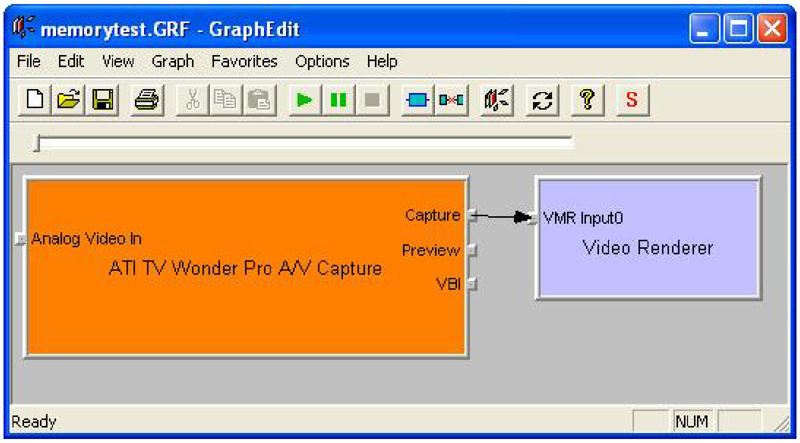


Figure 4-1 Graph Edit example

An example of using Graph Edit is shown in Figure 4-1. This Figure demonstrates the use of Graph Edit. In the example, two filters are inserted into the editor, a video capture filter and a render filter. After the two filters are connected, pressing the play button in the tool bar causes the graph to start playing. A video display Windows pops up and the captured video starts to play (Figure 4-2). By using Graph Edit, a quick TV application can be created in a matter of minutes without having to write a single line of code.

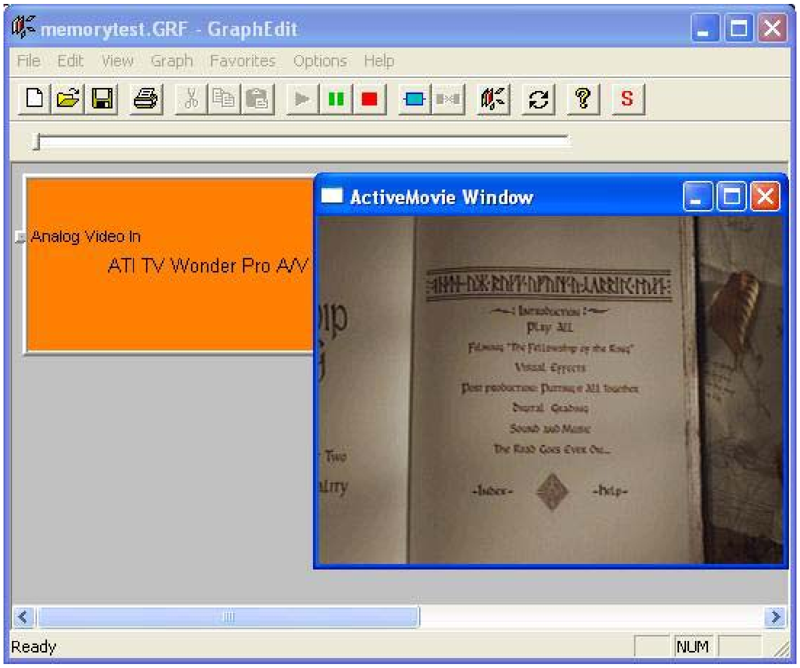
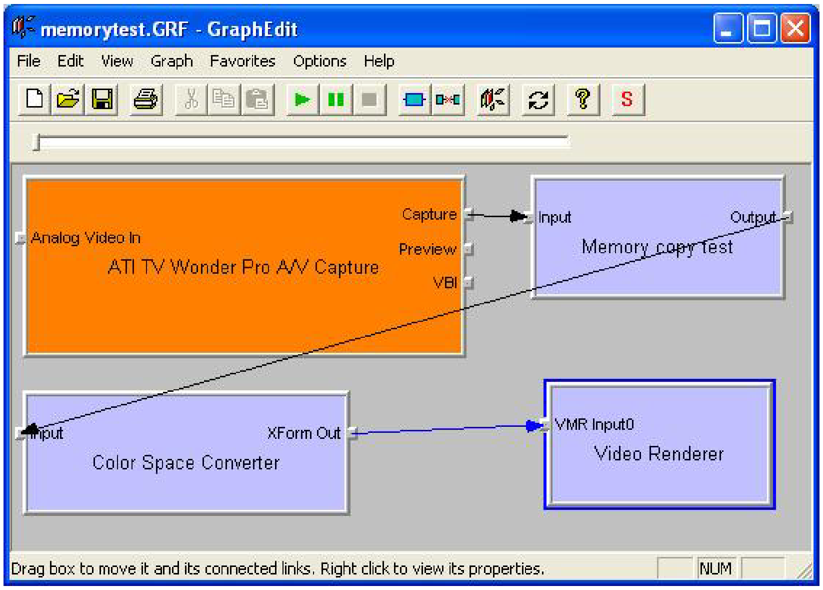


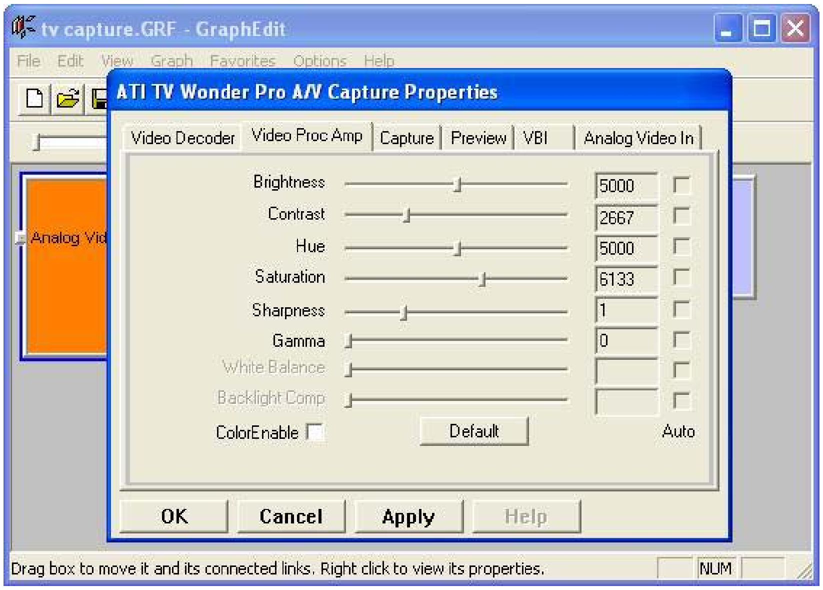
Figure 4-2 Graph edit playing example

In some graphs, intelligent connect is very useful. In Figure 4-3, initially there were only three filters in the graph, the capture filter, the memory copy transform filter and the video renderer. After connecting the capture filter to the transform filter, an attempt is made to connect the transform filter to the render filter. There is a problem with doing this. The transform filter uses a different video format then the render filter, and they are not compatible. The two filters initially fail when trying to connect. After this happens, Graph Edit uses intelligent connect to attempt to connect the two filters using any number of intermediate filters. In this case, Graph Edit automatically finds a fourth filter, which converts the format the memory copy filter provides into a format the video renderer desires. In this example, it is a simple conversion from 16 bit Red Green Blue (RGB) video to 32 bit RGB video. It is also possible to change the settings on each filter. This is shown in Figure 4-4, where it is possible to change the video color settings on the capture card.



Drag box to move it and its connected links. Right click to view its properties.

Figure 4-3 Memory copy test graph


tv capture.GRF - GraphEdit
File Edit View Graph Favorites Options Help
Analog Vid
ATI TV Wonder Pro A/V Capture Properties
Video Decoder
Video Proc Amp
Capture
Preview
VBI
Analog Video In
Default
OK
Cancel
Apply
Help
NUM

Brightness	5000	☐
Contrast	2667	☐
Hue	5000	☐
Saturation	6133	☐
Sharpness	1	☐
Gamma	0	☐
White Balance		☐
Backlight Comp		☐
ColorEnable ☐		Auto

Drag box to move it and its connected links. Right click to view its properties.

Figure 4-4 Changing the settings example

4.5 Filter Programming Example

Understanding how a filter works is not complicated, however, it is difficult to explain. To fully appreciate how a filter works, a simple example is presented that explains most of the basics of creating a filter and having it interoperate with other filters. This example is similar to sections in both the DirectX SDK [Micr02] and a reference book [Pesc03]. Sample code is used to show the key functionality of the filter. The sample code is taken from the actual memory copy test filter. Some of the functions, are shortened to eliminate some of code, that although is necessary for the filter to work, is not that useful in explaining how it works. Hence, all of the code can be considered pseudo code rather then the exact code that exists in the filter.

The filter given as an example is commonly referred to as a null filter or in this case a memory copy filter. This filter has no processing inside of it. All it does is receive data from the input filter, copy it, and then send it to the output filter. It can be used as a starting point for a more complex transform filter where the filter manipulates the data, or as shown later, streams the data over the network.

There is a lot more to creating filters then just the information provided in this section. Doing anything more complex then a simple null filter requires more functionality then is shown in this section. Even changing the compiler settings for the first time to get the filter to compile is quite time consuming. This section focuses on showing how a filter works through explaining its basic functionality, rather then providing exact instructions on how to create one.

4.5.1 Using the DirectShow SDK

Included with the DirectX Software Development Kit (SDK) is the DirectShow SDK. The SDK does not normally come with Windows and is a separate download from Microsoft's web site. It includes Graph Edit, sample projects, and documentation. The SDK also includes utility code that does a lot of the common low-level operations. This is very similar to the MFC, which provide much of the code for low level Windows interfacing that significantly speeds up development. The code included in the SDK is invaluable and saves a lot of programming time.

The memory copy filter is not created from scratch. The project uses three classes provided by the SDK: CBaseFilter, CBaseInputPin, and CBaseOutputPin. Writing a new class that inherits these base classes saves a lot of time. All of the common functionality that all filters have is already included in the base classes. Only functionality that is specific to the filter is required to be written.

The SDK also includes a very useful helper class called CTextMediaType. A DirectShow media structure, which contains all of the specifics of a media format is partially composed of unintelligible COM Globally Unique Identifier (GUID). This makes debugging difficult. A GUID is a 16 byte identifier that is different for every COM object. This class contains functionality to convert COM GUIDs to readable strings, and other functionality to convert the entire media structure to a readable string. An example of the output might be "Major Type:MEDIATYPE_Video – Sub Type: MEDIASUBTYPE RGB555 - Format: RGB 320x240, 16 bits" rather then a group of numbers.

4.5.2 The Memory Copy Filter

4.5.2.1 Basic Configuration

The first step when creating a filter is to have it compiled and registered by the operating system. The easiest way to do this is to copy pre-existing sample code included with the SDK. The important functionality to copy is the barebones that allow the filter to work within the COM framework. Several of the copied data structures have to be altered which change the characteristics of the filter.

The first is the COM GUID of the filter which has to be changed as it should be different for every COM object:

```
DEFINE_GUID(CLSID_JNULL,
0xe2035248, 0x895e, 0x4817, 0x8e, 0x7f, 0xac, 0x3a, 0x9d, 0xcb, 0xcc, 0x1f);
```

This GUID is generated from a Windows utility program called uuidgen.exe. Next, the filter's characteristic structures have to be customized to the null filter. These structures are used when the filter is being registered with the operating system so that the operating system knows what kind of filter it is, and in what category (audio, video, encoders, decoders etc) to place it. An example of the structures are shown below:

```
const AMOVIESETUP_MEDIATYPE sudPinTypes =
{ &MEDIATYPE_NULL       // clsMajorType
, &MEDIASUBTYPE_NULL };  // clsMinorType

const AMOVIESETUP_PIN psudPins[] =
{ { L"Input"          // strName
  , FALSE             // bRendered false unless rendering
  , FALSE             // bOutput true if pin is output
  , FALSE             // bZero true if filter can have zero instances of this pin
  , FALSE             // bMany true if filter can have more then one instance of this pin
  , &CLSID_NULL       // clsConnectsToFilter obsolete
  , L""               // strConnectsToPin obsolte
  , 1                 // nTypes number of media types supported
  , &sudPinTypes      // lpTypes
  }
, { L"Output"         // strName
  , FALSE             // bRendered
  , TRUE              // bOutput
  , FALSE             // bZero
  , FALSE             // bMany
  , &CLSID_NULL       // clsConnectsToFilter
  , L""               // strConnectsToPin
  , 1                 // nTypes
  , &sudPinTypes        // lpTypes
  }
};

const AMOVIESETUP_FILTER sudNullNull =
{ &CLSID_JNULL                // clsID
, L"Memory copy test"               // strName
, MERIT_DO_NOT_USE              // dwMerit
, 2                         // nPins
, psudPins };
```

Once the filter is compiled for the first time, to register the filter with the operating system, that is to make it available to be loaded through Graph Edit, the program regsvr32.exe is used. Several other functions are required for COM and the registration processes. However, they are common to all filters and can be copied from sample code. Since they do not add to the understanding of how filters work, they are not included in this discussion.

4.5.2.2 The Main Filter Class

The main class for the filter is the one derived from *CBaseFilter*, called *CFilter*. Overriding a base class makes *CFilter* rather simple with only a few functions required in order to work. The three functions required have to do with configuring the filters input and output pins. The first function is the constructor at which the input and output pins are created.

```
m_pInputPin = new CInputPin ( NAME ("CInputPin"),this, & m_crtFilter, phr, L"Input" ) ;
m_pOutputPin = new COutputPin ( NAME ("CInputPin"),this, & m_crtFilter, phr, L" Output " ) ;
```

CinputPin and *COutputPin* are described later in the chapter. The next two functions are rather important to understanding how the filter works.

```
int GetPinCount ()            { return 2 ; }

CBasePin GetPin (IN  int Index)
{
   CBasePin *pPin ;
```

```
  if (Index == 0)
     pPin = m_pInputPin;
  else if (Index == 1)
     pPin = m_pOutputPin;
 else
    pPin = NULL;
  return pPin ;
}
```

The preceding code is simple but it allows Graph Edit to recognize the pins of the filter. The code is used internally in the *CBaseFilter* class to enumerate all of the filter's pins. It allows for the internal enumeration functions to know how many pins are in the filter and to get access to all of them by repeated calling of the *GetPin* function with higher index numbers. The enumeration function is the one called by Graph Edit to Figure out how many pins the filter has and how to display them.

This is a good example of good software engineering. It is useful to put as much of the functionality into the base class as possible. In this case the mechanism that Graph Edit uses to enumerate all of the pins of the filter is put into the base classs. The base class can contain all the functionality, but it must be guaranteed that the inheriting class will somehow provide the specific details of its pins. This is done by the base class requiring the inheriting class to override certain functions, functions that provide it with the necessary data. These functions are referred to as "pure virtual" functions. That means that the inheriting class (the one that inherits all the functionality of *CBaseFilter*, in our case *CFilter*) is required to create its own versions of the pure virtual functions for the class to compile. Because the base class knows that the child class is forced to override these functions, it can use the functions without worrying whether the functions exist, or if the pin data is available.

The *CFilter* class requires one more function to be useful. The function listed below is shortened, as the actual *Receive* function has more code in it, but this function shows basically what the function does. It is called by the input pin when it has data ready, then makes a copy of the data and sends the copy to the output pin:

```
HRESULT CNullNull::Receive (IN  IMediaSample * pms)
{
     pms = Copy(pms);
     hr = m_pOutputPin->Deliver(pms);
     return hr;
}
```

This section explains how a filter is created and how the filter is recognized by Graph Edit. All of the key functionality, in this case just the data copying would occur in the main filter class. If the filter had to do any transformations it would call a function such as *Transform(pms)* that would transform the data before sending it to the output pin. What is missing from the filter is that even before the filter can receive a media sample, it must negotiate the connection type both with the input and output filter. This leads to the following discussion of what the input and output pins looks like.

4.5.2.3 Input/Output Pins

The Input pin class *CInputPin* inherits from the *CBaseInputPin* class and the Output pin class *COutputPin* inherits from the *CBaseOutputPin* class. Once again, the base classes provide much of the functionality, including the process by which the pins negotiate connections. The base classes require the child classes to override several pure virtual functions to provide the base classes with the information of what media types the pins require/provide.

The negotiations start when Graph Edit attempts to connect two filters together. The process begins and the output pin looks at which media types are preferred by the input pin. A media type data structure looks like:

```
typedef struct  _MediaType {
   GUID      majortype;
   GUID      subtype;
   BOOL      bFixedSizeSamples;
   BOOL      bTemporalCompression;
   ULONG     lSampleSize;
   GUID      formattype;
   IUnknown  *pUnk;
   ULONG     cbFormat;
   [size_is(cbFormat)] BYTE *pbFormat;
} AM_MEDIA_TYPE;
```

After the media type is identified as video by looking at the *majortype* GUID, the *pbFormat* pointer can be type cast into the video information header structure. This is done in C code by doing this:

```
VIDEOINFOHEADER *pVIHeader = (VIDEOINFOHEADER *) pbFormat;
```

This is telling the compiler to fill a *VIDEOINFOHEADER* (shown below) structure with the data contained in pbFormat.

```
typedef struct tagVIDEOINFOHEADER {
   RECT              rcSource,
   RECT              rcTarget;
   DWORD             dwBitRate;
   DWORD             dwBitErrorRate;
   REFERENCE_TIME    AvgTimePerFrame;
   BITMAPINFOHEADER  bmiHeader;
} VIDEOINFOHEADER;
```

The *bmiHeader* also contains more useful information.

```
typedef struct tagBITMAPINFOHEADER {
   DWORD  biSize;
   LONG   biWidth;
   LONG   biHeight;
   WORD   biPlanes;
   WORD   biBitCount;
   DWORD  biCompression;
   DWORD  biSizeImage;
   LONG   biXPelsPerMeter;
   LONG   biYPelsPerMeter;
   DWORD  biClrUsed;
   DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

These structures and the corresponding ones (not shown) for audio generally define every format available. By using type casting, only one base structure is required for every media type. This simplifies function calls, as there is only one variable parameter required to define all media types. The variable actually used is of type *CMediaType*, which is a wrapper class. A wrapper class is a class that inherits all the data of a structure, in this case the *AM_MEDIA_TYPE* structure, but also provides additional functionality that is useful.

The functions that the base classes use to find out which media types are suitable during the negotiations are as follows:

```
HRESULT GetMediaType (IN  int iPos,  OUT CMediaType *pmt )
HRESULT  CheckMediaType (IN  const CMediaType *  pmt)
```

The *GetMediaType* function returns the preferred media type for a given pin, given an index identifying the pin. This function is used by the base class to enumerate all of the media types by going through the index from zero to whenever the function returns a value that specifies that no more media types exist.

The *CheckMediaType* function is used to confirm that a given media type is acceptable by the pin. An example negotiation might go as follows. The output pin calls the input pin's *GetMediaType* to see what media types the input pin prefers. If any of them match, by having the output pin return true when calling its own *CheckMediaTy,e* then there is a match. If there is not a match the output pin can enumerate its own preferred types by calling its own *GetMediaType* and calling the inputs pins *CheckMediaType* to see if any of them are acceptable. If there are no matches then the two filters cannot directly connect.

For the null filter, the input pin will accept any format the connecting filter's output pin requires, since all the filter does is copy the data and send it to the next filter.

```
HRESULT  CInputPin::CheckMediaType (IN  const CMediaType *  pmt)
{
        m_mt.Set(*pmt);
        return S_OK;
}
```

The function also saves the media type into the *m_mt* variable, as it is used in the null filters output pin when negotiating that connection. Since the null filter does not do any transformations, the media samples produced by this filter are in exactly the same format as they came in. When the output pin negotiates its format it can only accept the format that the input pin agreed to. The output pins *CheckMediaType* function might look something like this:

```
HRESULT COutputPin::CheckMediaType (IN  const CMediaType * pmt)
{
    HRESULT hr ;

   CMediaType InputType;
   CInputPin *pInputPin = (CInputPin *)m_pFilter->GetPin(0);

   // get information from the input pin
   pInputPin->GetMediaType(0,&InputType);

   if (*pmt == InputType)
   return S_OK;

  return S_FALSE;
}
```

The only thing the code does, is to check if the input pin that the output pin is trying to connect to, has the same media type as the output pin requires. If it does, this function returns true. Note that the if statement is pseudo code and in reality it would be required to check different parts of the media structure to see if the two are alike enough to warrant a match.

After the pins have negotiated an agreed media type, the pins need to find a way to transport the data from the output pin to the input pin. The most common transport is local memory transport which simply uses the main computer memory. Note, that there are hardware options, such as a video capture card, that directly transfers data to a video card bypassing main memory. In most cases, the output pin decides what the size of the memory buffer should be. A common example of this is when a video stream that is 320 pixels wide and 240 pixels high at 24 bits per pixel, would require a buffer of at least

320x240x3 = 230400 bytes. The output pin allocates a buffer of this size and then sends the buffer information to the input pin to verify that it is the correct size.

The buffer allocator negotiations are mostly handled by the base classes, but they require several functions to be implemented in the child pin classes.

```
HRESULT CInputPin::NotifyAllocator(IMemAllocator *pAllocator, BOOL bReadOnly)
```

The *NotifyAllocator* function is implemented by the input pin to verify that the allocator settings are valid. The *IMemAllocator* interface contains all of the functionality that allows the input pin to get the properties of the buffer.

```
HRESULT COutputPin::DecideBufferSize (IN  IMemAllocator *   pIMemAllocator,  OUT ALLOCATOR_PROPERTIES
*  pProp)
```

The *DecideBufferSize* function is implemented by the output pin to initially decide on the buffer size and actually create the buffer. Now that the filters can be connected, they now have to be able to transfer the data. The input pin contains a function called:

```
HRESULT CInputPin::Receive (IN  IMediaSample * pIMediaSample )
```

All media in DirectShow use the *IMediaSample* to send samples from one filter to another. *IMediaSample* provides the interface to the media sample. Recall that a COM interface is a collection of functions with no direct access to data. In order to access the data, the data must be obtained through function calls. The *IMediaSample* interface is more or less a wrapper interface that contains a means of accessing the raw data through function calls.

```
HRESULT GetPointer(BYTE **ppBuffer);
LONG GetActualDataLength(void);
```

GetPointer returns a pointer to a byte array, which contains the media sample data. This function requires another function, *GetActualDataLength* to return the length of the byte array. Other important information is given by the interface using the functions,

```
HRESULT GetTime( REFERENCE_TIME *pTimeStart, REFERENCE_TIME *pTimeEnd);
HRESULT IsSyncPoint(void);
```

GetTime returns the start and end time of a sample in a 100 nanosecond time scale. For video this would be the start and stop time when a sample should be displayed. Keep in mind that all video is, is a continuous stream of bitmaps being displayed at a constant rate. *IsSyncPoint* returns true if the media is a synchronization point. A synchronization point is a media sample that does not require a previous sample to play correctly. All audio samples are automatically synchronization points. Some video encoder use temporal encoding. This encoding compresses video frames by having a key frame that contains all of the image data. Frame following the key frame are prediction frames which uses the previous frame data and information on the data that changed to generate the subsequent frame. A key frame is a synchronization point.

The *COutputPin* class contains a function called *Deliver* that is used to transfer the data to the next filter. The *Deliver* function is implemented by the base class.

```
HRESULT Deliver(IMediaSample *pSample);
```

4.5.2.4 Summary of Creating a Filter

A filter can be created by first creating the structures and variables and copying over the base code required for COM objects. Then the filter classes *CFilter*, *COutputPin*, and *CInputPin* are created by inheriting base classes that are provided in the SDK. Any functions that are pure virtual in the base classes must be overridden with the appropriate code. At this point, the filter can be compiled and be used to simply transfer data. Although no functionality is added to the null filter example, simple transforms are easy to add (such as changing the contrast of the video).

This section covers the basics of creating a simple transform filter. The other types of filters, Source and Renderer, are created slightly differently using different base classes and require extra functionality. Even though there are more complex filters, this section covers all the basics that are required for every type of filter, such as initial creating, pin media type and allocator negotiation, and data transfer.

Chapter 5 Media Streaming Architecture

This section presents the general architecture of the media streaming server and client. The media includes both audio and video. The general architecture of the media streaming software is shown in Figure 5-1. The media streams are captured, encoded, packetized and then buffered into the server. The server is responsible for creating the connection between the server and the clients. If there are any clients connected to the server, it sends out the media packets to the clients. The clients place all the packets into buffers, at which point the packets are combined into their original form. A periodic check occurs that verifies whether all the packets have been received correctly. If any are missing, the client can issue resend requests to the server. The media is then synchronized (audio and video synchronization) and sent to the decoders, which passes the media to the renderers.

There are three components to the overall architecture, the server, the client and the transport layer. The server includes everything used to capture the video streams as well as any connection and streaming functionality. The client includes receiving completed media samples, synchronizing them and having them play. The transport layer is the part which receives media samples from the server, forms packet for the server, receives the packets at the client end, recombines them and makes them available for rendering. Because the transport layer is a major component of both the server and client and entails most of the interesting work in the thesis, it is considered its own component.

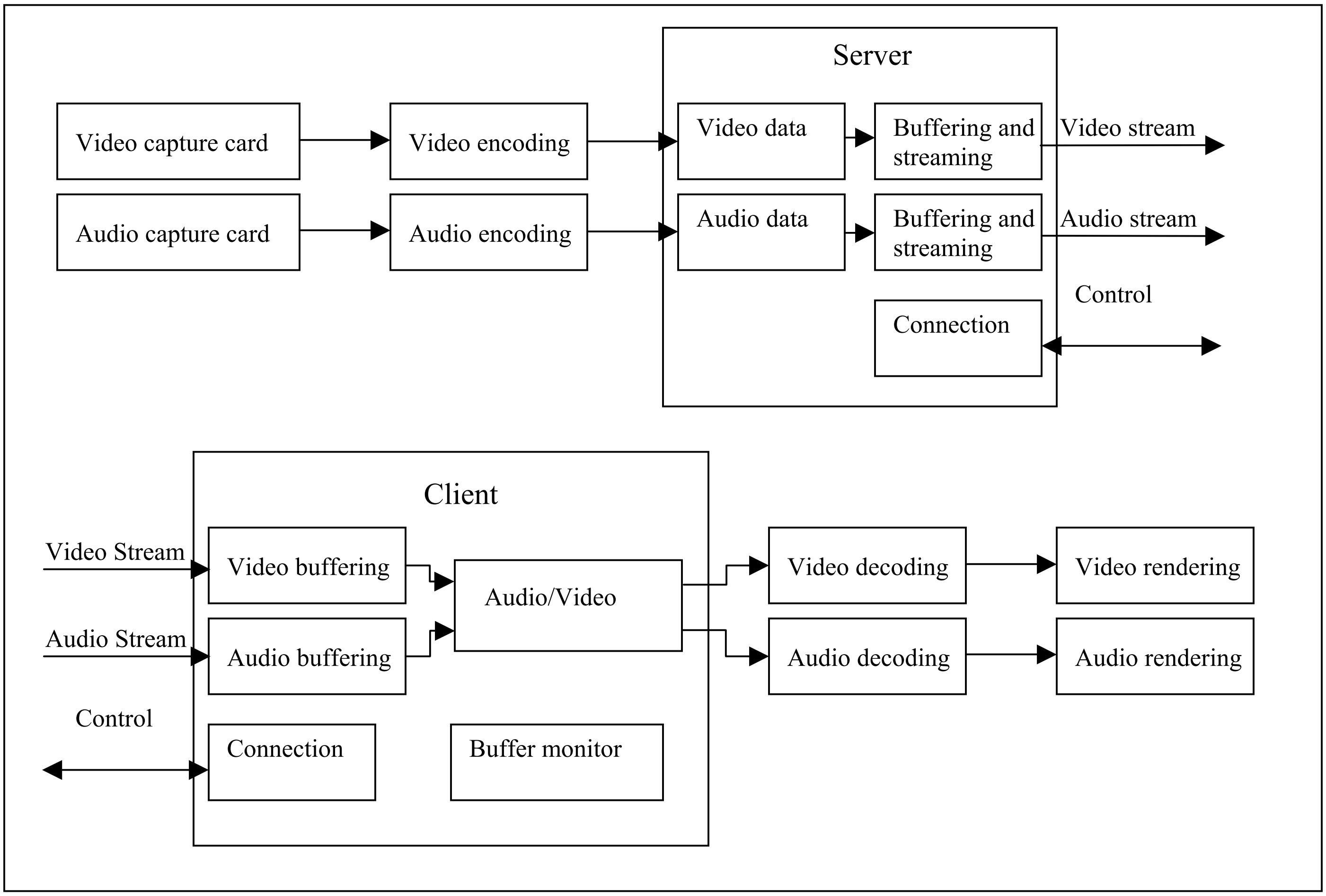


Figure 5-1 General architecture

5.1 The Transport Layer

The media streaming application is designed to evaluate the transport layer. The purpose of the transport layer is to have the network connection between the server and client be as close as possible to having the video and audio capture cards directly connected to the client computer and streaming media directly to the video and audio cards. Factors that evaluate how well the transport layer simulates a direct connection include the latency, bit rate and frame loss probablity.

If a live media stream is created on a local machine its latency is near zero. Video is normally at 30 frames per second and this translates into a frame every 33ms. That means that for stream to be continuous without missing any frames, the processing time on average has to be less then 33ms. On a local machine the bit rate can be very high because all data transport is done in local memory or on hardware buses both of which have very high bandwidths. The frame loss on a local machine should be zero as nothing should be lost.

When transporting data over the network various problems may occur. Latency can be relatively high compared to local memory transport. However, the latency of a wireless network is no where near the high latency of the Internet. The bandwidth of a wireless network is also limited compared to a local machine. There is also the potential for frame losses over any kind of network including in particular a wireless one. The desired result of the transport layer is to best simulate the local memory transport. The transport layer should prioritize compensating for the frame loss by having it as close to zero as possible. Next, the latency should be low, but not so low as to not have enough time to request a frame resend if one gets dropped from the network. To maintain high broadcast quality, the transport layer should use up as much of the network bandwidth as possible. However, it should not use all of it, to allow for any necessary resend request and control messages that must be sent.

The transport layer can be thought as a virtual local memory transport. It should seamlessly transfer the data across the wireless link and reproduce a synchronized media stream, albeit with higher latency. To do this the transport layer requires a packetizer, a packet header format, server, and client buffering and synchronization.

5.1.1 Software Architecture of the Transport Layer

5.1.1.1 Header Format

Each media sample is received from the DirectShow filters in the form of an *IMediaSample*. *IMediaSample* is an interface that is used to pass any kind of media across DirectShow filters. The interface contains functions to obtain all the data that is contained in the media sample. The interface is also versatile as it has functionality and options to include all the information that any possible media sample requires. Although there are more fields contained in *IMediaSample* then shown below, the following discusses fields that are relevant to our discussion.

1. The actual data: This is the video or audio media sample (it is compressed).
2. The data length: The length of the video or audio media sample.
3. Start and stop times: The time in units of 100 nanoseconds of when the media sample should be played. This is used for synchronization.
4. Synchronization point: A flag that is enabled if the media sample is a synchronization point. That is if it is a compressed video frame, it is a synchronization point if it is a key frame. All audio frames are synchronization points.

A media sample can be any size. There is an ideal size that the media sample should be packetized. Since the data is being sent as UDP packets, the maximum packet size should be the maximum size of the payload of a UDP packet. Because most video samples are larger than the maximum packet size, the sample has to be split up into many packets and then recombined into the original sample by the receiver. In addition to all the information that is obtained from the media sample, the following additional information should be stored. (This is very similar to real time protocol [Perk03] except the session ID is not required):

1. Packet identifier: This identifies that this packet is a media stream packet.
2. Packet index: The index of the packet (each frame increments the index by one) so it could be placed in the buffer at the correct place.
3. Check sum: This can be used to detect packet errors but used mostly for debugging as UDP has its own error checking.
4. Current packet fragment: Since media samples may be split into many packets this stores the fragment index so all of the fragments can be correctly recomposed into the original media sample.
5. Total number of fragments: This indicates how many fragments compose the entire media sample. It is used mostly to detect when a media sample is missing some of its fragments.
6. Resend flag: If a particular packet is being resent because it did not arrive the first time at the client this flag is set.

5.1.1.2 The Packetization Process

The process of packetization is as follows. The packetizer receives the media sample. It then obtains all of the information from the media sample as required (start and stop times, etc…) and does all other preprocessing such as generating the checksum. It then separates the sample into as many packets as are required. Each packet has its own header that contains the complete information about the media sample. The server periodically checks if any packets are available, and if so it sends them out in turn. The client then receives the packets and has to recombine them.

The entire process is not that complex but the use of good software engineering practices can simplify the implementation of the process significantly. The packetizing process is very similar to the depacketizing process. To take advantage of this, both the client and server use the same packetizing class. It is actually composed of two classes for modularity sake. The header class has all the functionality to generate and parse the headers for each packet. The packetizer class has all of the functionality to both generate and recombine packets. The packetizer class inherits the header class, and then the packetizer class is inherited by the buffer class, whether it beoutgoing or incoming (they are different as the incoming buffer is more complex). For this to work, both the outgoing and incoming buffers use the same data structure as defined in the packetizer class to form their circular buffers. This is shown in Figure 5-2.

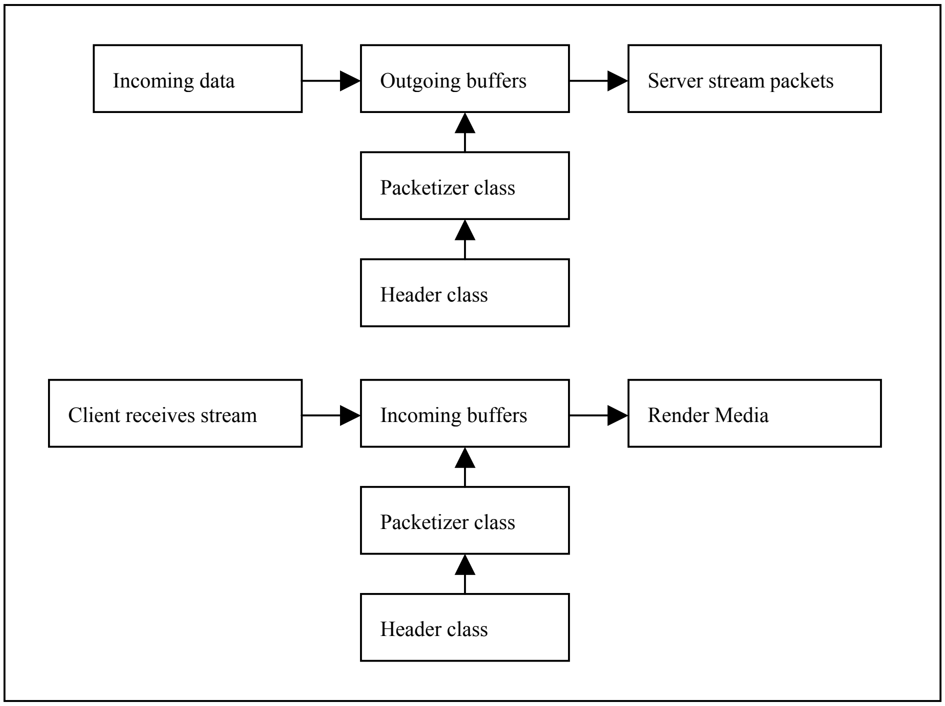


Figure 5-2 Packetizing class

This approach has the following advantages:

1. The same code is used in both the server and client thereby speeding up the development process.
2. The packetizer class contains both the code for packetizing and recombining, which means it can be tested separately of any network connection to see if the packetizing/recombining process maintains data integrity. If there are any bugs in

the code, they are discovered during the development of this class, rather then when the data is being sent across a network, which would be far more difficult to debug.

3. Any changes to the header format are made much easier by the modular architecture. All of the header information is added by the header class as it is required, so only a few code changes can change the header format for both the client and the server. Other classes are oblivious to the changes and do not need to be altered in any way.

The actual process of packetizing the data is explained below. Note that some information on how the buffering works is oversimplified and explained later in more detail.

1. The media sample is received, included with the data is the data length, start and stop times and synchronization point.
2. The outgoing buffer allocates a data block to store the packets, and calls:

```
bool Packetizer::InsertData(BYTE *Data, int Length,
                            LONGLONG Start,  LONGLONG End, int IsSyncSource,
                            Packetizer::DATABLOCK &DataBlock)
```

3. First the relevant data is added to internal variables in the header class using function calls:

```
void rtp_SetSyncSource(int IsSyncSource);
void rtp_SetPacketSize(int PacketSize);
etc (more set function calls)
```

4. Then the data is separated into appropriately sized data chunks with each chunk, now a packet, getting its own header by making the following function call:

```
int rtp_AddHeader(BYTE *&buffer);
```

5. The packets are stored, in order, in the same data block in a contiguous space in memory and are now ready to be sent out.
6. The server periodically checks to see when data is available to send and sends out the packets as soon as possible.
7. The client receives the data, verifies it is a media packet by checking the header.
8. The packet is then placed/recombined into the buffer with a call to:

```
Packetizer::EnterFragment(BYTE *Msg, int Length,Packetizer::DATABLOCK &DataBlock)
```

9. This function uses the header class to parse out all of the header information
10. The header class parses out the header and stores the information in internal memory variables that can be retrieved using the function calls:

```
int rtp_ParseHeader(BYTE *&buffer);
void rtp_GetSyncSource(int &IsSyncSource);
void rtp_GetPacketSize(int &PacketSize);
etc (more get function calls)
```

11. When all of the fragments of a packet have arrived and have been recombined the media sample is ready to be sent to the decoder.

5.1.1.3 Outgoing Buffer

The buffering used in the server and client is required for the recovery of lost packets. The buffer size, in terms of the number of media samples, must be large enough to account for the time it takes for the client to recognize a lost packet, send a resend request and receive the missing packet. The optimum size of the buffer is not important in the architecture discussion, but, the buffers must provide a way to increase in size if required.

Normally, the outgoing buffer would not need to be large because it can be assumed in a LAN environment that the data packets would be sent out nearly immediately. If there are any problems, such as congestion for more then a small time period, this would cause much larger problems in terms of overall bandwidth use, so this situation will not be discussed here.

The outgoing buffer is required by the resend mechanism. If the client request that a packet be resent, the server must have the packet available somewhere, otherwise it would not be able to resend it. The outgoing buffer is implemented as a circular buffer. Each data block in the buffer contains information on the packets it is storing, which makes it easy to find the packet required for a resend. When loading and retrieving packets from the circular buffer, the read and write indexes are constantly changing. At no time is the data actually deleted. The buffers data is only overwritten when the write index loops around, the buffer. As long as the buffer has not looped around, the packet will still be available for a resend. The outgoing buffer size should be made appropriately large, so the buffer will not over write any packets that may be required to complete a resend request.

The outgoing buffer is fairly simple with the only complexity being fulfilling out resend request. Key functionality is as follows:

```
void InsertData(IMediaSample *pSample)
bool GetNextPacket(BYTE *Data, int &Length);
void IncrementNextPacket();
void ResendPacket(int TotalPacket, ULONG PacketIndex, int FragmentIndex);
```

InsertData is the function that is used to insert a media sample into the buffer. *GetNextPacket* is used to get the next packet to send if one is available. If the packet has been successfully sent, *IncrementNextPacket* is called to increment the buffer. *ResendPacket* is called if the server has received a resend request. It handles the search of the entire buffer for the missing packet. To prioritize resent packets, the outgoing buffer actually consists of two buffers, one for resent packets and the other for new packets. The resent packets are given priority because it is deemed that packets that are being resent require more timely delivery than new ones.

5.1.1.4 The Incoming Buffer

The incoming buffer is more complex than the outgoing buffer. It is responsible for recombining media samples from packets and issuing resend requests if any are missing. A packet is entered into the buffer with a call to the *OnReceive* function:

```
void OnReceive(BYTE *Data, int Length);
```

This function receives the packet as a byte array. The first thing the function does is call the header function *rtp_ParseHeader*, after which the packet and fragment index can be obtained. If all the packets are received in order without any losses, all the function has to do is enter the packet into the next available data block. However, since packets can be lost, there might be empty spaces in the buffer that might later be filled with packets from a resend request. Because of this, packets must be entered by their packet index.

The algorithm that finds where to place the packet is as follows:

1. A search occurs of the entire circular buffer to see if the received packet's index already exists somewhere in the buffer.
2. If it does exist, it knows where to place the packet.
3. If it does not exist, it takes a look at the last packet in the buffer, and based on the packet index of the last entry, it can Figure out where to place the new packet.

4. If the new packet is the next packet counting up from the last packet that is entered, the spot is found.
5. If the new packet is not the next packet, rather than allocate the next available spot, its spot in the circular buffer is allocated according to an offset. The offset is calculated as the current packet index subtracted from the last packet index in the buffer.
6. Any block in the buffer between the previous last packet and the new packet must be correctly initialized to its proper packet index, so that when the resend request occurs those packets have a place to be entered.
7. Once the correct spot has been determined, the *EnterFragment* function is called from the packetizer class to enter the packet into the buffer.

There has to be some sort of mechanism to check that the buffer is being filled correctly and to issue any resend requests if there are missing packets. The function that does this is called *VerifyIntegrity*. The function uses the following algorithm:

1. Check every buffer entry between the read and write indexes of the circular buffer.
2. If an entry has zero size, it means that that particular packet was never received. It has been only entered in the buffer because a packet with a higher index was inserted further down the buffer. In this case the entire packet is missing. Send a resend request for the entire packet.
3. Otherwise, if the buffer entry the function is checking is the last one in the buffer. Check that all the fragments in the packet have been received up to the last one that is entered. The packet might not be complete but everything up to the last fragment that

is entered should exist. If any in-between packet fragments are missing, send resend request only for these fragments.

4. Otherwise, for every other buffer that does at least partially exist, and is not the last one, the function knows that the entire packet with all its fragments should be complete. If the buffer entry is missing any fragments, send a resend request for that fragment.

In all instances, before sending a resend request ,the algorithm checks a countdown timer. The *VerifyIntegrity* function is run asynchronously at a high rate. A countdown timer on individual buffer entries prevents resend request from being sent for the same packet every time the *VerifyIntegrity* is run. After a resend request is issued, a countdown timer is set to a specified amount that is larger that the time required for the resend request to be fulfilled by the server (so bandwidth would not be wasted on repetitive request). Also, to compensate for the possibility of out of order packets, the countdown timer could be set by default to a certain amount of time. This would mean that a resend request would not be immediately issued for a missing packet to give it an opportunity to arrive which could be useful if there is a possibility of out of order packets.

The incoming buffer makes a best effort attempt to always containing the next available media sample. The function that is used to obtain the next sample is:

```
bool GetNextBuffer(DATABLOCK *&DataBlock);
```

This function returns the next available media sample in the buffer.

There are more complex resend request mechanisms possible. One example [JaEl98] checks if the retransmission attempt would have enough time to reach the client. It is also possible to prioritize resends of the video key frames which are far more important then the normal video frames. If the resend request does not arrive in time to be played, the entire video frame is dropped. This is done to prevent the decoder from quitting due to an error. In general, depending on the decoder, there can be some resilience when part of the data is missing [Tall98]. However, due to the way a frame is compressed, if one byte is missing the entire frame may be lost.

5.2 Server Architecture

The server application contains the following functionality:

1. Interface to the DirectShow filters to obtain media samples and change settings.
2. The ability to receive media samples from the DirectShow capture streams.
3. A mechanism to allow clients to connect to the server and register to receive the media streams.
4. Ability to packetize, buffer and send outgoing media streams.
5. Contain a GUI to display various statistics on the live streaming process.

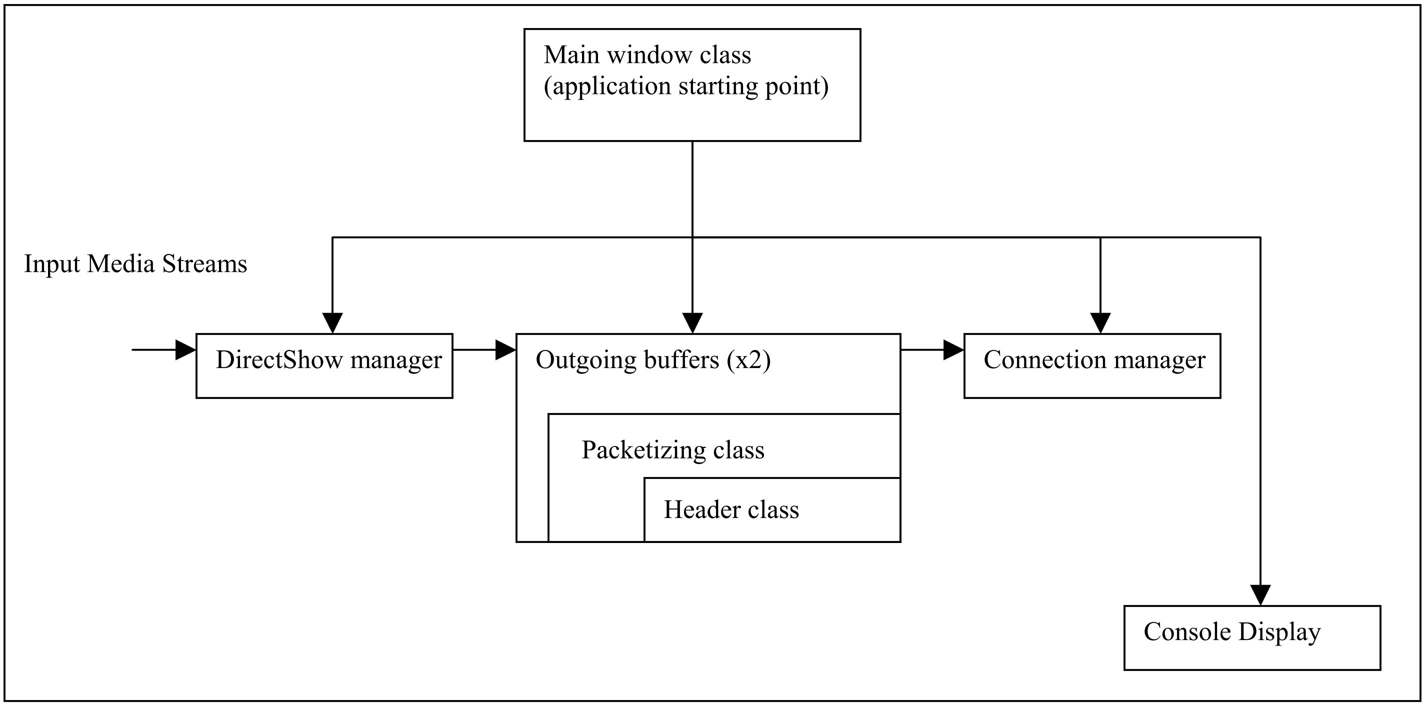


Figure 5-4. Server architecture

Figure 5-4 shows the software architecture of the server. The architecture uses a modular design. Each class has its own distinct functionality. This means each class is not too large or complex, and is easily modifiable. Overall there are five main C++ classes, DirectShow Manager, Connection Manager, Console Display, Outgoing buffer and the Main Window. The DirectShow manager handles all interfacing with the DirectShow filters, including receiving the media streams. The Outgoing buffer is described in the previous section. The Connection Manager provides all network transport functionality. This includes everything involved with setting up the connections with the clients and sending out the media streams. The Console Display is used for debugging and displaying status messages. The main window class is necessary in Windows programming as it starts up the application and receives all user event messages.

5.2.1 The Main Window Class.

All applications that are programmed for Windows contain a Main Window class. This is the class that is the first one created when the program is run. It therefore is responsible for initializing all other classes. The Main Window class is also the one that receives all of the event messages. In Windows, asynchronous events (e.g clicking on a mouse, key strokes, socket receiving a message, etc…) are routed to a window event handler and cannot be directly sent to an arbitrary class. The Main Window class is responsible for receiving these messages. Examples are the start and stop stream messages, and the adjust bit rate messages.

Usually it is desirable to minimize any functionality placed in Main Window class. It is tempting to put much of the functionality into this class because it is the one that receives the event messages and it seems easier to simply put the event functionality into this class. However placing too much functionality, especially functionality from different parts of the program usually leads to a confusing mess of code. The only real functionality that the Main Window is given is to gather statistics and paint them on the screen. The Main Window is the one that receives repaint event messages and it has access to all of the other classes, therefore it is the logical place to put the statistics drawing functionality. Otherwise, the main window class is kept simple. It contains event handlers, most of which offload the functionality to other classes, initialization procedures and statistics gathering and painting routines.

5.2.2 The DirectShow Filters

The server through the DirectShow manager, loads up a filter graph that streams the media to the DirectShow manager. It is simpler to use Graph Edit to make the filter graph and then load it up, rather then constructing it by code. The filter graph used is shown in Figure 5-3.

The filter graph uses the SoundMax Digital Audio 0001 filter to capture the audio and then sends it to the LAME Good Edition MP3 encoder. This is then sent to the SendFilter. On the video side, the ATI TV Wonder Pro is used to capture the video. It is sent to the Microsoft Windows Media 9 filter, which encodes it into WMV (Microsoft's MPEG-4 format). This is then sent to the SendFilter. The filter graph also contains a preview window, which is useful for previewing the video stream. It also contains two additional filters, SoundMax Digital Audio and ATI TV Wonder Pro Tuner. These can be used to alter some of the preprocessing settings, such as input volume and determining which channel the tuner is capturing.

The SendFilter is a custom filter that is used to send the streamed media to the DirectShow manager. The filter is very similar to the memory copy test filter explained earlier. The functionality of the SendFilter is to connect to the audio/video encoders and, upon receiving media samples from them, to pass the samples to the DirectShow manager. SendFilter does not contain any other processing. To pass the samples, the SendFilter uses callback functions. A callback function is when a function is maintained as a pointer, and it can be called just like a normal function. In this case, the DirectShow manager passes to the SendFilter two callback functions, one each for audio and video. These functions are used to pass the samples.

To do this SendFilter has the following two functions:

```
STDMETHODIMP CSendFilter::SetCallback( SAMPLECALLBACK Callback )
{
   m_callback = Callback;
   return NOERROR;
}

STDMETHODIMP CSendFilter::SetCallbackA( SAMPLECALLBACK Callback )
{
   m_callbackA = Callback;
   return NOERROR;
}
```

Note that functions and variables with an extra “a” are the audio version as there are two streams each with the same functionality but with different names. *m_callback* and *m_callbackA* are of type:

```
typedef HRESULT  (*SAMPLECALLBACK)(IMediaSample * pSample);

CALLBACK m_callback;
CALLBACK m_callbackA;
```

The callback functions have an argument of type *IMediaSample* which, as explained before, contains all of the media data required. These function are used to pass the media samples as shown below:

```
HRESULT CSendFilter::Receive ( IN  IMediaSample * pms )
{
   if (m_callback != NULL)
   {
          // use the call back function
          m_callback(pms);
   }
   return S_OK;
}

// audio version of receive
HRESULT CSendFilter::ReceiveA ( IN  IMediaSample * pms )
{
   if (m_callbackA != NULL)
   {
          // use the call back function
          m_callbackA(pms);
   }

   return S_OK;
}
```

The SendFilters, *Receive* and *ReceiveA* are called in a similar way as *Receive* is called in the memory test filter. The video input pin calls the *Receive* function when it has received a media sample, and the audio input pin calls the *ReceiveA* function when it has received a media sample.

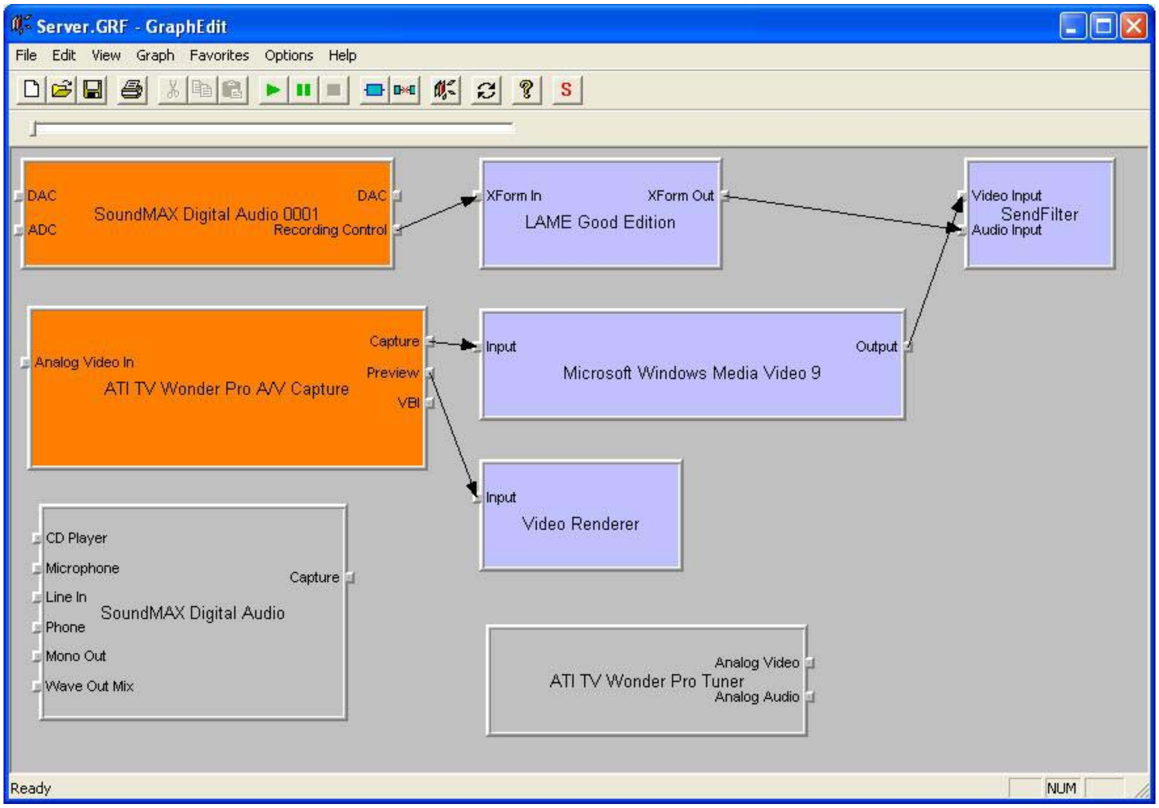


Figure 5-3 Server filter graph

5.2.3 The DirectShow Manager

The DirectShow manager handles all of the functionality to interface to the DirectShow filters that create the media stream. Included in which is the custom filter, *SendFilter*, that enables the media streams to be captured and then sent to the DirectShow manager. Two functions that are of interest are the initialization function and the change bit rate function. The first thing the DirectShow manager does is to runs its initialization function *Init*:

```
 void IDirectShow::Init()
{
  // start up com
  hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);

  if (hr != S_OK)
  {
         AfxMessageBox("Warning cannot start up COM");
         return;
  }

hr = CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void
**)&m_pGraph);

  if (hr != S_OK)
  {
         AfxMessageBox("Warning cannot initialize the filter graph");
         return;
  }

  hr = LoadGraphFile(m_pGraph,L"c:\\profiles\\Server.GRF");

   m_pGraph->FindFilterByName(L"SendFilter",&m_pGrabber);
   m_pGraph->FindFilterByName(L"Microsoft Windows Media Video 9",&m_pEncoder);
   m_pGraph->FindFilterByName(L"ATI TV Wonder Capture",&m_pAtiCapture);
   m_pGraph->FindFilterByName(L"Video Renderer",&m_pPreview);
   m_pGraph->FindFilterByName(L"ATI TV Wonder TV Tuner",&m_pTVTuner);
   m_pGraph->FindFilterByName(L"LAME Good Edition",&m_pAEncoder);

  IGrabberBuffer *pGrabber;
  hr = m_pGrabber->QueryInterface(IID_IGrabberBuffer,(void **)&pGrabber);
  hr = pGrabber->SetCallback(AppCallback);
  hr = pGrabber->SetCallbackA(AppCallbackA);
  pGrabber->Release();
}
```

The *Init* function starts up by linking to COM, which is required before using COM. It then creates an instance of a filter graph and loads up the interfaces of the filter. The interfaces are stored for later use, as some are useful in configuring the properties of the media stream. The *IGrabberBuffer*, which is the interface to the custom *SendFilter* is loaded up and two functions *SetCallback* and *SetCallbackA* are called.

The callback mechanism is not entirely straight forward because call back functions are not entirely compatible with c++. When using c++ functions, to call them both, the function and the member class have to be known. To work around this problem, static functions that are not members of the class are used as the callback functions and they call the class's member functions. This is shown in the following code where a pointer to the member class is used to call the member function. All the member function does is insert the media sample into the outgoing buffer, which handles all of the packetizing:

```
// static call back allows it to latch on to the grabber filter
HRESULT __stdcall AppCallback(IMediaSample * pSample)
{
   return pIDirectShow->DataReceived(pSample);
}

HRESULT __stdcall AppCallbackA(IMediaSample * pSample)
{
   return pIDirectShow->DataReceivedA(pSample);
}

HRESULT IDirectShow::DataReceived(IMediaSample *pSample)
{
   g_OutputBuffer.InsertData(pSample);
   return S_OK;
}

HRESULT IDirectShow::DataReceivedA(IMediaSample *pSample)
{
   g_OutputBufferA.InsertData(pSample);
   return S_OK;
}
```

The DirectShow manager also has a mechanism to change the bit rate of the video stream. This is done through a function called *ChangeBitRate*:

```
void IDirectShow::ChangeBitRate(int RateIndex)
{
    m_pIControl->Stop();

    BYTE PTempState[2976];
    int StateNum = 2976;

    char FileName[100];
    if (RateIndex == 0)
            sprintf(FileName,"c:\\profiles\\enc80.prf");
    else if (RateIndex == 1)
            sprintf(FileName,"c:\\profiles\\enc85.prf");
    else if (RateIndex == 2)
            sprintf(FileName,"c:\\profiles\\enc90.prf");
    else if (RateIndex == 3)
            sprintf(FileName,"c:\\profiles\\enc95.prf");
    else
            return;

    LoadState(PTempState,StateNum,FileName);
    hr = m_pEncControl->SetState((void *)PTempState,StateNum);

    m_pIControl->Run();
}
```

Although there are several generic COM interfaces that allow for the bit rate to be changed, the video encoder used provides only one that works. The pointer to the interface *m_pEncControl* allows preset profiles to be loaded into the encoder. This interface is created in the *Init* function. Eight such profiles are created that have various different quality settings. These have different bit rates and key frame rates. *ChangeBitRate* simply loads up the desired profile and loads it into the encoder. Note that the function stops the media stream and then restarts it. Also when changing the bit rate the client must be notified to tell it to flush its buffers, stop and re-start.

5.2.4 The Connection Manager

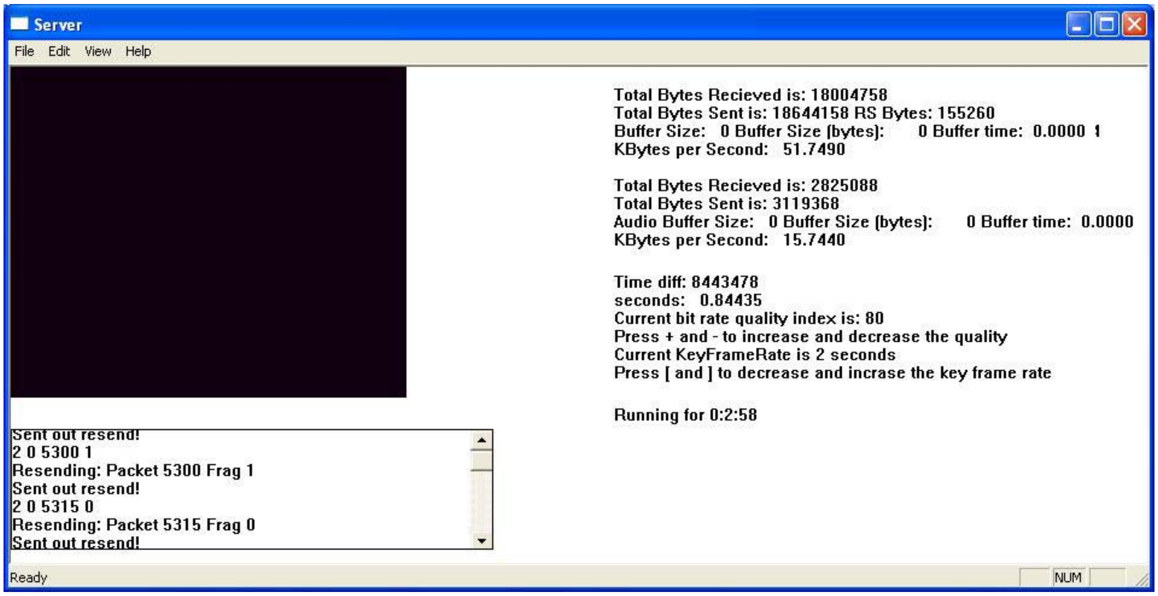
The connection manager handles all of the networking. It uses UDP sockets exclusively. It has one socket and also the corresponding event handler *OnQueryRecieved,* that receives all incoming connection and resend requests. When a client requests a connection, the client IP address is added to the list of clients. When each client connects they are allocated their own streaming sockets. This approach is found to be more reliable under congestion conditions due to each socket having its own internal buffer preventing overflow. The connection manager has two worker threads, one for each stream, which constantly check the outgoing buffers for any packets and if it finds any, it sends them out to the clients.

5.2.5 Outgoing Buffers

There are two buffers used to send the data, one for the audio and one for the video. They operate exactly as explained in the transport section.

5.2.6 GUI, Console and Statistics

The GUI of the server is shown in Figure 5-4. The GUI contains a preview window of the video stream, a console that displays status messages, and various statistics. The only controls available are to start and stop the stream, and to change the video quality and key frame rate, thus lowering or increasing the bit rate.



```
Total Bytes Recieved is: 18004758
Total Bytes Sent is: 18644158 RS Bytes: 155260
Buffer Size:   0 Buffer Size (bytes):      0 Buffer time:  0.0000  1
KBytes per Second:   51.7490

Total Bytes Recieved is: 2825088
Total Bytes Sent is: 3119368
Audio Buffer Size:   0 Buffer Size (bytes):      0 Buffer time:  0.0000
KBytes per Second:   15.7440

Time diff: 8443478
seconds:   0.84435
Current bit rate quality index is: 80
Press + and - to increase and decrease the quality
Current KeyFrameRate is 2 seconds
Press [ and ] to decrease and incrase the key frame rate

Running for 0:2:58
```

```
Sent out resend!
2 0 5300 1
Resending: Packet 5300 Frag 1
Sent out resend!
2 0 5315 0
Resending: Packet 5315 Frag 0
Sent out resend!
```

Figure 5-4 Server GUI

5.3 Client Architecture

The Client contains the following functionality:

1. The ability to send a media stream to the DirectShow decoders and renderers.
2. The ability to receive media samples from the server, and form them from packets into media samples.
3. The ability to buffer the samples and request resends if any are missing.
4. The ability to synchronize the video and audio streams.
5. A GUI to display various statistics on the live streaming process.

The functionality of the client is essentially the inverse of what the server does. The architecture of the client, shown in Figure 5-4, looks very similar to the architecture of the server. The only real difference is that the client receives media streams from a network and then sends them to the DirectShow filters, whereas the server does the opposite. Several classes are reused between the server and the client (the packetizer, header and console) and the rest are similar. The most important difference not discussed so far is the synchronization mechanism, which will be discussed in detail. Most of the other classes are similar to the server's so they will not be described in as much detail.

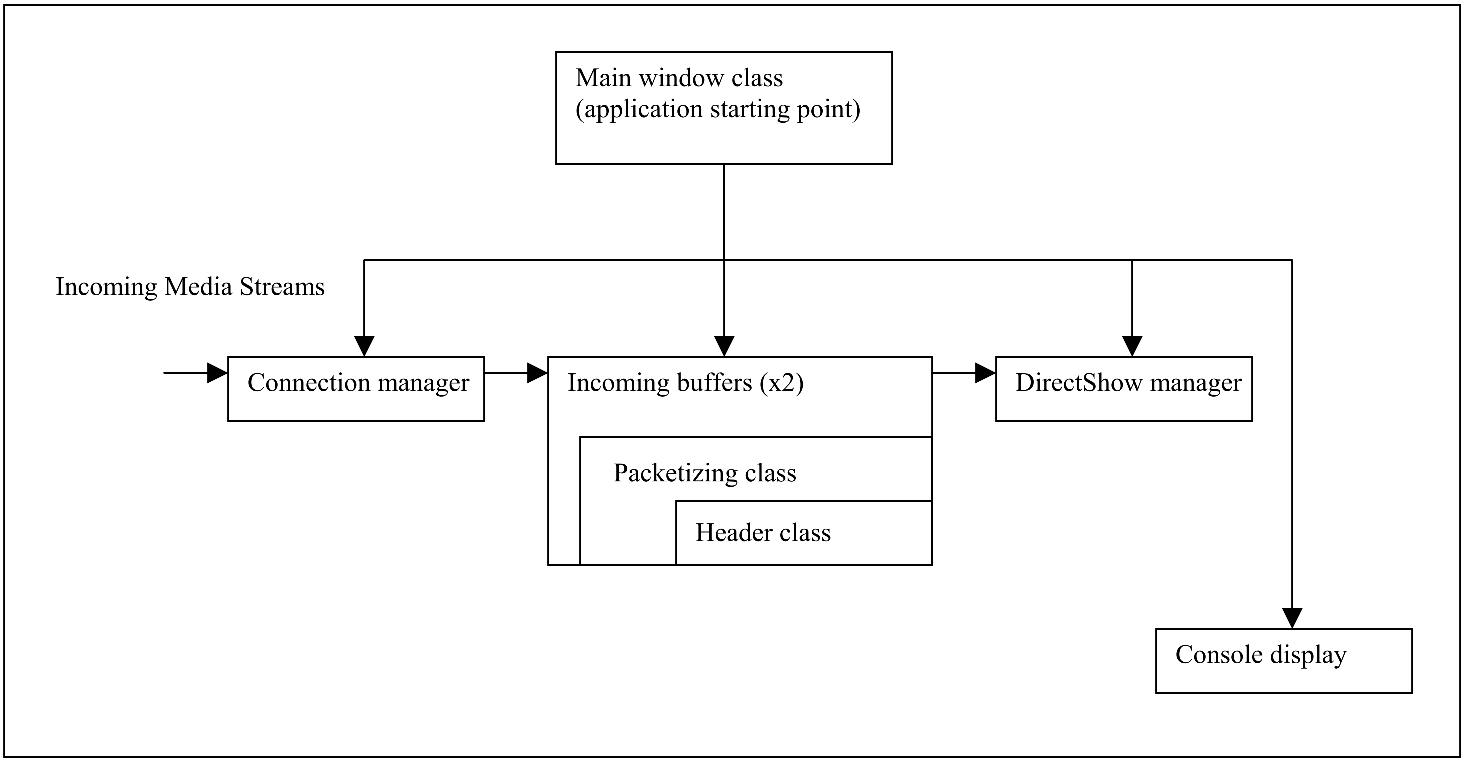


Figure 5-4 Client architecture

5.3.1 The Main Window Class/Connection Manager.

Since the connection manager for the client is rather simple, its code is included in the Main Window class. The only connection based responsibilities for the client are to notify the server that it requests certain media streams, to send resend requests, and to receive the media streams. When incoming data is received, it triggers an event handler which routes the data into the incoming buffers. There are two threads, one to handle each of the audio and video. These take data out of the buffers and send them to the DirectShow filters to be played.

5.3.2 The DirectShow Filters

The client also loads up the filter graph from a file. There are five filters in the graph as shown in Figure 5-5. Network Receiver is a custom filter that receives media samples from the client applications, and sends them to the decoders. WMVideo Decoder DMO decodes the video and MPEG Layer-3 Decoder decodes the audio. The Full Screen Renderer is used to display the video. To debug, a windowed renderer can be used as it does not take up the entire screen. The Audio Filter is a custom filter designed with low latency that replaces the generic Audio Filter that is available as part of DirectShow.

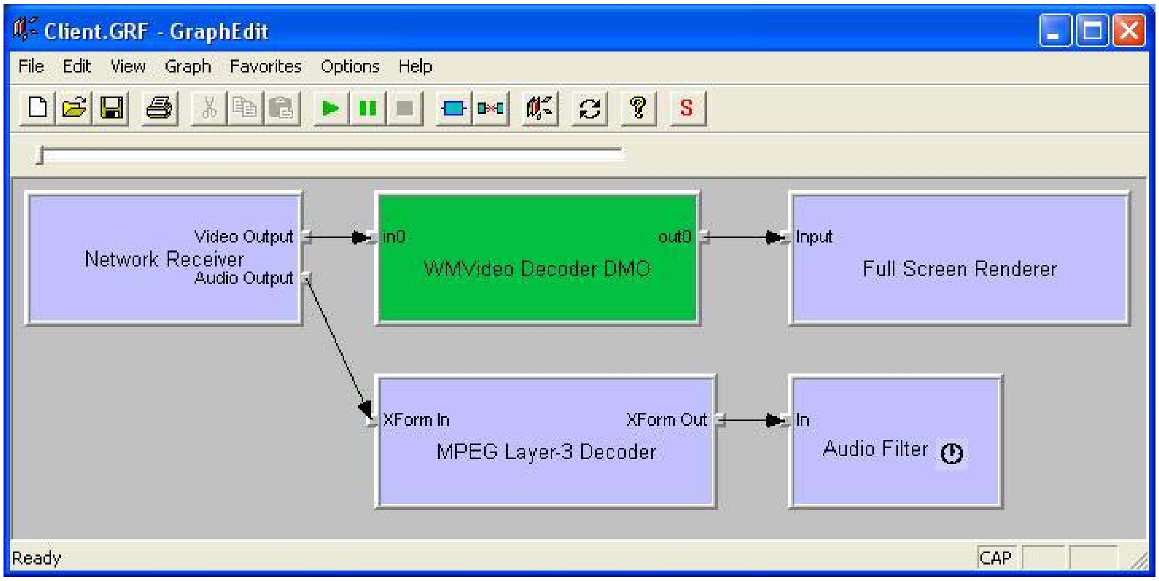


Figure 5-5 Client graph

5.3.2.1 Media Synchronization

The Network receiver is considerably different from the memory copy example. The main difference is that it is a source filter. The source filter has quite a few extra responsibilities. Each output pin of a source filter represents a separate media stream. In this case we have two pins one for the audio one for the video. Each media stream is considered running in its own thread. A media sample appears at the output pin and is sent to the upstream filters. After the sample is finished being rendered, control returns to the source filter to stream another sample. This means that the Network Receiver is responsible for the timing of each stream.

The Network Receiver is also responsible for synchronizing the audio with the video streams. The client sends it video and audio samples that are already closely timed. However, the timing is not perfect. The client cannot do the timing exactly because it does not have any control over the hardware buffers. The main problem with synchronization is that the default Audio Filter has an underdetermined amount of buffering. To solve this, the custom Audio Filter is created with a set and measurable amount of buffering.

The video is easy to synchronize because the video is a collection of bitmaps being played continuously. It is very easy to speed up or slow down the video. However, it is more difficult to speed up audio, as some sound processing has to occur to accomplish this. To slow down audio it is possible to delay playing a sample, but that might leave noticeable gaps in the audio.

The process used to synchronize the audio and video is based on matching the video to the audio in a way that is similar to [LiZa04]. The audio is played continuously with the video being adjusted to match the audio. The custom audio filter returns the time stamp of the audio being currently played. The video time clock can then be set to this time. This allows the video to either be delayed or sped up to match the audio. This allows the video to be within one sample time of the audio. This leads to what appears to the user to be perfect synchronization.

5.3.2.2 The Network Receiver

The Network Receiver is the source filter for the filter graph. Some of its processing is similar to that of the memory copy filter. The way it connects to other filters is identical to the memory copy filter. The big difference between the two is in the way the network receiver has to synchronize the media streams. This functionality is provided through Network Recievers custom interface:

```
HRESULT STDMETHODCALLTYPE PushMedia(IMediaSample *pSource) = 0;
HRESULT STDMETHODCALLTYPE PushMediaA(IMediaSample *pSource) = 0;
HRESULT STDMETHODCALLTYPE GetLastTimeStamps(LONGLONG &Video, LONGLONG &Audio) = 0;
HRESULT STDMETHODCALLTYPE GetBufferSize(int &VideoSize, int &AudioSize) = 0;
HRESULT STDMETHODCALLTYPE GetCurrVClock(LONGLONG &Time) = 0;
HRESULT STDMETHODCALLTYPE SetCurrVClock(LONGLONG Time) = 0;
HRESULT STDMETHODCALLTYPE FlushBuffers() = 0;
```

To send the media to the filter there are two functions, *PushMedia* and *PushMediaA*. The first one is sent video samples and the second one audio samples. These samples are then sent to the output pins to be transferred to the connecting filters. Because the filter is a source filter, the filter inherits the *CSource* base class and the output pins inherit the *CSourceStream* class. The *CSourceStream* class has a built in thread which is used to stream the media samples. Most of the threading functionality is hidden inside the base class. To expose the thread, the *FillBuffer* function is used. This function is a pure virtual function and must be implemented. When the graph is initially run, the *FillBuffer* function is called. This allows the output pin to literally fill the buffer with a media sample and have the sample sent to the connecting filter. When the media sample has been sent all the way to the renderer, and the renderer is done playing, the *FillBuffer* function is called again. In this way media samples can be continuously played. This process is shown in Figure 5-5.

DirectShow has an internal mechanism to synchronize the video and the audio to a single clock. It uses an internal clock, typically the audio card's timer or the high resolution timer. Usually the video sample includes the start and stop time. The internal DirectShow clock would have the video play for the specified time, and then return to the *FillBuffer* function once the samples end time has elapse. The audio would be using the same clock. Thus, ideally, using the internal mechanism, the two media streams would be synchronized. The only problem is that the clock is not designed for live streaming and it is difficult to work with. As well there is insufficient documentation on how to use the clock. Because of this a custom clock is created using the high resolution clock in the computer.

The operation of the clock is based on the two functions *GetCurrVClock* and *SetCurrVClock*. The client takes a look at the last audio sample played and then compares the sample's time with the time of the last video sample. If they are significantly different (more then one frame out) the video clock is set to the time of the last audio sample played using *SetCurrVClock*.

Since the internal DirectShow clock is not used, the video timing must be controlled. This is done by the following lines of code, located inside the *FillBuffer* function. The *Sleep* command pauses the function until the specified time has elapsed.

```
g_JClock.SetTimers(0,StopTime);
```

```
while (!g_JClock.CheckTimer(0))
{
        Sleep(1);
}
```

The audio does not require a clock. The audio samples represent a continuous audio sound rather then a collection of bitmaps to be displayed at certain time intervals. To make use of this, the custom Audio Filter is used, and the Network Receiver simply sends the audio samples without any delay as soon as they are available to the connecting decoder filter.

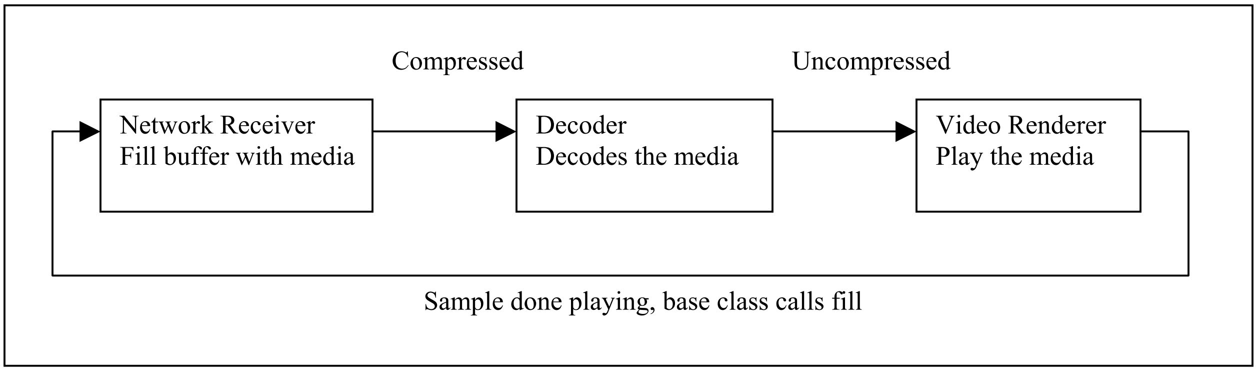


Figure 5-5 Playing example

5.3.2.3 The Audio Filter

Pulse Code Modulation (PCM) is used as the audio format inside of a computer. The audio capture card measures the amplitude of the audio at fixed intervals and converts these into binary values. To play audio on a computer, the PCM data has to be loaded into the sound card's memory buffer. There are two ways to stream audio into the sound card. One is using a streaming buffer, the other is using a static buffer.

A static buffer is used to play sound for a short duration. The PCM data is loaded into the sound buffer, and then the sound card is told to play the content of the buffer.

The buffer is played once. For streaming sound a streaming buffer is used. The streaming buffer contains a read and a write point. The buffer is continuously filled at the write pointer, while it is being played at the read pointer. This kind of buffer is often used for music or other continuous audio. Because the buffer cannot be infinitely large, it is implemented as circular buffer with built in looping. (It is interesting to note that when a computer locks up while playing music, the music seems to loop continuously even though the computer stopped working. The reason for this is that a streaming buffer is used and the sound card is looping through the buffer rather than stop playing it.)

The advantages of a streaming buffer is that for continuous sound, the sound will not be disjointed as it might be with a static buffer. Using a static buffer, the buffer is loaded, played, stopped and then must be loaded again. If there are any delays in the system in writing to the buffer they can be noticeable. Using a streaming buffer, however, means that some audio data needs to be pre-buffered before the buffer can be played. This adds to the latency in the system. One of the reasons for creating a custom Audio Filter is that the latency can be kept to a minimum of one sample being pre-buffered.

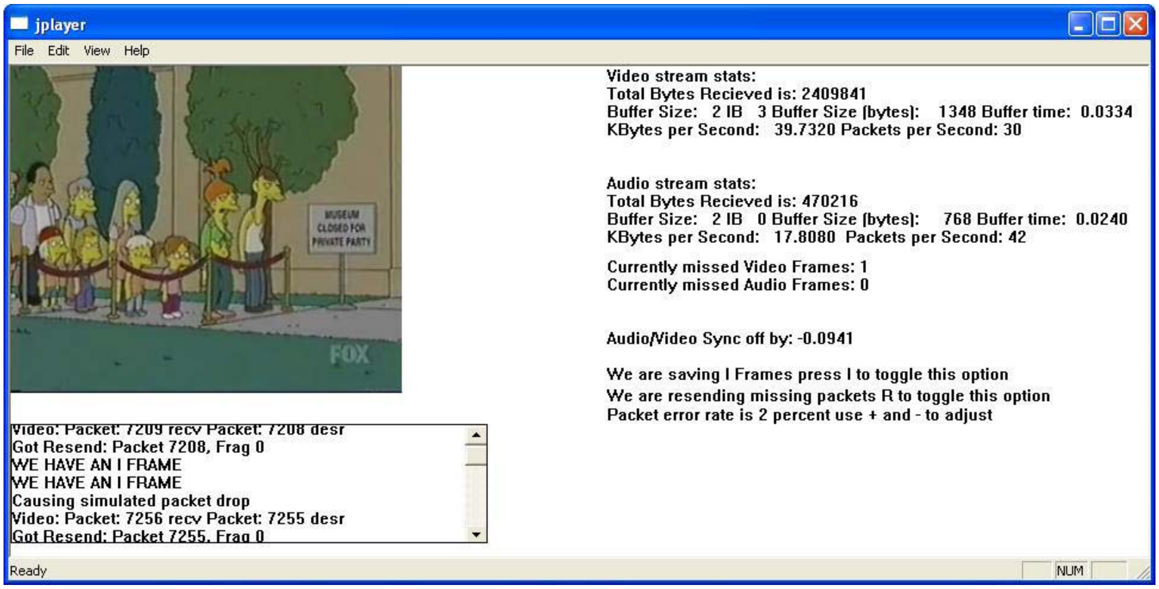
The Audio Filter itself is quite simple. It inherits *CBaseRenderer*. A render filter is much simpler than the other types because all it has to do is play a media sample, and it is restricted to one pin. As usual, the base class handles most of the work with the only two functions required to be overridden being *DoRenderSample* and *CheckMediaType*. The base class includes the input pin with the *CheckMediaType* pure virtual function. This being the only noticeable inclusion of having the input pin combined with the main class. The *DoRenderSample* does all the work. It receives a media sample to be played. An abbreviated version of this function is shown below. Note that it is pseudo code and is missing some details. Note the “if” statement towards the end where the sound buffer only starts playing once there are enough samples in the sound buffer:

```
HRESULT CAudioFilter::DoRenderSample(IMediaSample *pMediaSample)
{

   HRESULT hr;

   // get sample information
   DWORD Length = pMediaSample->GetActualDataLength();


   BYTE *pData;
   pMediaSample->GetPointer(&pData);

            NumSamples++;

   // copy in the sample
   m_pSoundBuffer->Lock();
   memcpy(lpvWrite, pData, Length);
    m_pSoundBuffer->Unlock
   if (!m_bIsPlaying && (m_NumSamples >= m_nBuffSamples))
   {
          m_bIsPlaying = true;
           hr = m_pSoundBuffer->SetCurrentPosition(0);
           hr = m_pSoundBuffer->Play();
   }

            return S_OK;

}
```

The rest of the functionality of the filter concerns initializing the DirectSound device and buffer to use the sound card and the ability for the filter to track timestamps and return them using the custom interface method *GetTimeStamps*. The code to do this is not that interesting and is not shown.

Generally speaking, losing audio packets is quite bad. When this occurs various things can happen such as no audio for a given point in time, repeating audio, etc… Some techniques to mask the missing audio have been developed [PeHH98] but their use is beyond the scope of this thesis.

5.3.3 GUI and Statistics



```
Video stream stats:
Total Bytes Recieved is: 2409841
Buffer Size:   2 IB   3 Buffer Size (bytes):    1348 Buffer time:  0.0334
KBytes per Second:   39.7320 Packets per Second: 30


Audio stream stats:
Total Bytes Recieved is: 470216
Buffer Size:   2 IB   0 Buffer Size (bytes):    768 Buffer time:  0.0240
KBytes per Second:   17.8080  Packets per Second: 42

Currently missed Video Frames: 1
Currently missed Audio Frames: 0


Audio/Video Sync off by: -0.0941

We are saving I Frames press I to toggle this option
We are resending missing packets R to toggle this option
Packet error rate is 2 percent use + and - to adjust
```

```
Video: Packet: 7209 recv Packet: 7208 desr
Got Resend: Packet 7208, Frag 0
WE HAVE AN I FRAME
WE HAVE AN I FRAME
Causing simulated packet drop
Video: Packet: 7256 recv Packet: 7255 desr
Got Resend: Packet 7255, Frag 0
```

Figure 5-6 The player

Figure 5-6 shows the client window. In the figure the client is set to have a 2 percent packet drop rate. Due to the packet drop rate the resend mechanism can be observed in the console. The Client has options to change the packet drop rate, to save or not save "I" frames (key frames), and to disable the resend mechanism.

Chapter 6 Wireless Video Streaming Experiments

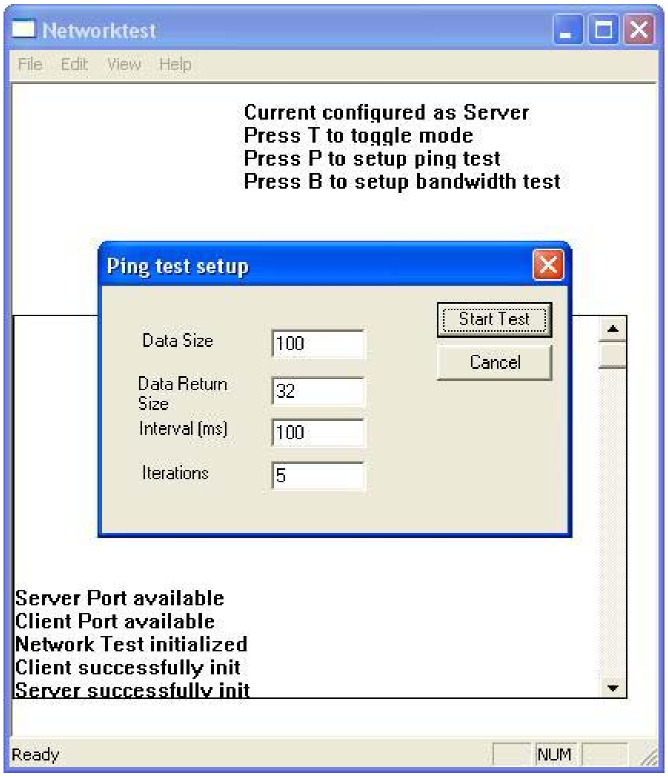
This chapter describes experiments with using the media streaming software as a test bench over two hypothetical scenarios on a standard computer network. The chapter starts by introducing one more application, the network test utility. This utility is then used to test the properties of different kinds of networks. To make it easier for others to use the software, instructions are given on how to install the client and the server. Also included, is a description of the statistics file generated by both the server and the client. Finally, the most important part of the chapter, the two test scenarios are presented.

The two test scenarios look at the two main networking issues, insufficient throughput and a packet loss rate. The first scenario is what happens when there is insufficient throughput for a short period of time due to a spike in the bit rate of the video. The second scenario examines what happens to the video quality when there is a packet loss rate, measured by the loss of whole video frames. The two scenarios are by no means meant to be a definitive test of different network connections, rather an example of the testing potential of using the media streaming software on a given network.

6.1 The Network Test Utility

To test the capacities and properties of the network, a network test utility is created. This is similar to other commercially available utilities that test network capacities and ping speeds such as Chariot [Athe03b]. The reason for creating a custom utility is that the socket code used is nearly identical to the socket code in the media streaming application. So it can be used to directly correspond to the media stream performance. Also it is modifiable and relativity easy to create.

The network test utility has two modes, Ping mode and Bandwidth test mode. It can also be switched between client and server so only one executable is required. There are two modes in the network test utility. The first is Ping mode. It has four options. The first is the packet size that is send out in the ping test. The second is the packet size that is returned in the ping test. Third is interval which is the time between successive pings, and forth is iterations which is is how many pings. The modes are shown in Figure 6-1. It is important to specify the data size because in a resend request, the request itself is small but the actual resend could be the maximum packet size.



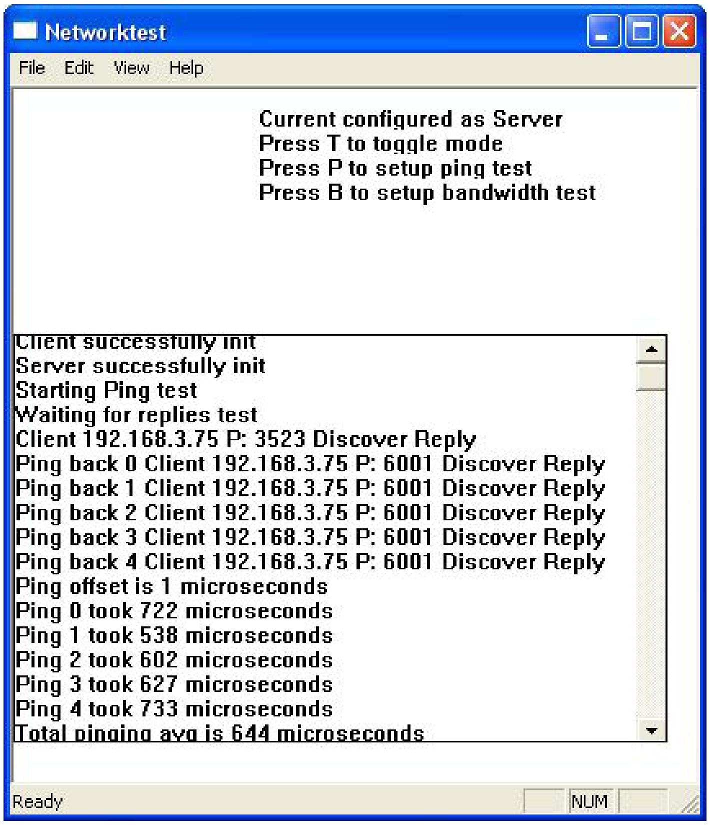
Current configured as Server
Press T to toggle mode
Press P to setup ping test
Press B to setup bandwidth test

Data Size	100
Data Return Size	32
Interval (ms)	100
Iterations	5

```
Server Port available
Client Port available
Network Test initialized
Client successfully init
Server successfully init
```

Figure 6-1 Ping mode

Figure 6-2 shows the ping test in operation. Looking at the console, the server discovers the client on the network by broadcasting out querying for a client and then pings the client. The pings are returned and the times are stored and displayed and then averaged. Note that for this test and all subsequent test there is no other user generated traffic on the network and few if any programs running in the background.

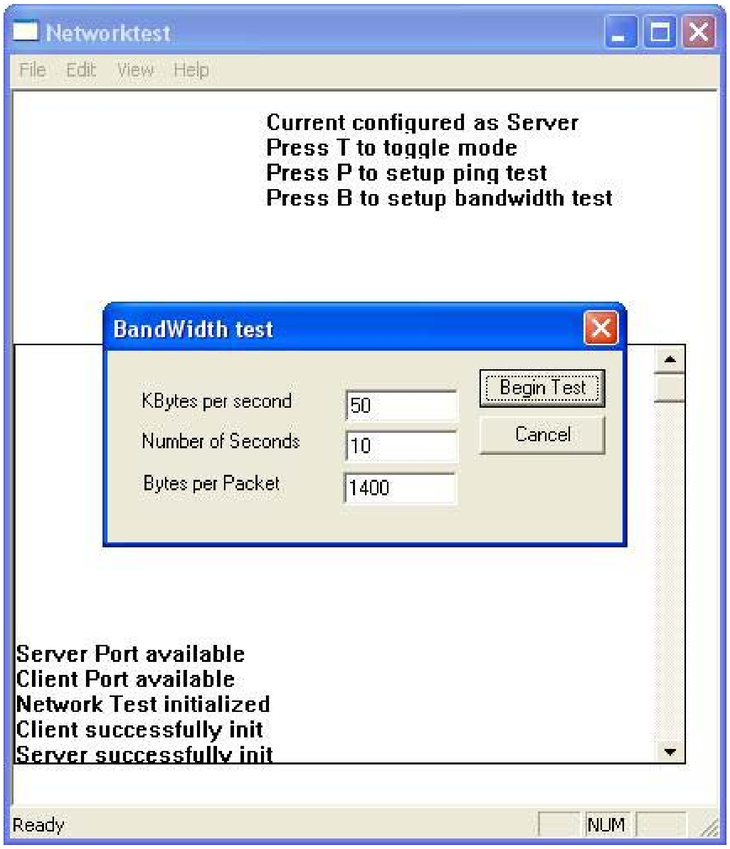


```
Current configured as Server
Press T to toggle mode
Press P to setup ping test
Press B to setup bandwidth test
```

```
Client successfully init
Server successfully init
Starting Ping test
Waiting for replies test
Client 192.168.3.75 P: 3523 Discover Reply
Ping back 0 Client 192.168.3.75 P: 6001 Discover Reply
Ping back 1 Client 192.168.3.75 P: 6001 Discover Reply
Ping back 2 Client 192.168.3.75 P: 6001 Discover Reply
Ping back 3 Client 192.168.3.75 P: 6001 Discover Reply
Ping back 4 Client 192.168.3.75 P: 6001 Discover Reply
Ping offset is 1 microseconds
Ping 0 took 722 microseconds
Ping 1 took 538 microseconds
Ping 2 took 602 microseconds
Ping 3 took 627 microseconds
Ping 4 took 733 microseconds
Total pinging avg is 644 microseconds
```

Figure 6-2 Ping test operation

The bandwidth test setup is shown in Figure 6-3. There are three options. Kbytes per second is the Kilobytes per second of traffic that the server sends out. Number of seconds is the number of seconds of the test, and bytes per packet is the number of bytes sent out per packet. This can be changed to simulate different packet sizes (for example audio packets that are smaller then the maximum UDP packet size).



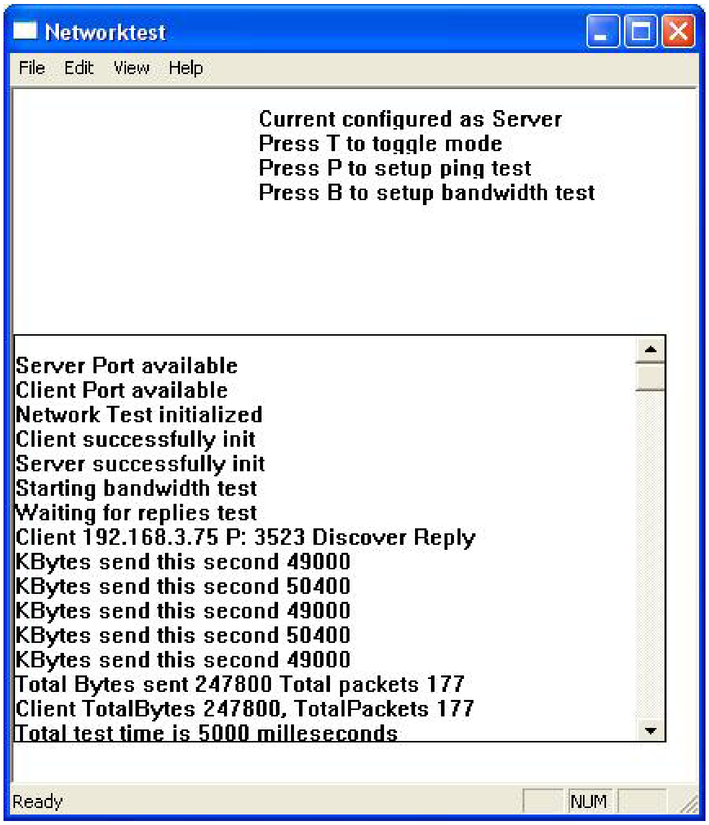
```
Current configured as Server
Press T to toggle mode
Press P to setup ping test
Press B to setup bandwidth test
```

KBytes per second: 50
Number of Seconds: 10
Bytes per Packet: 1400

```
Server Port available
Client Port available
Network Test initialized
Client successfully init
Server successfully init
```

Figure 6-3 Bandwidth test setup

Figure 6-4 shows a bandwidth test in operation. The server attempts to send over 50 Kbytes per second or as close to it as possible while keeping to the packet size requirements. After the test is complete, the server verifies how many bytes and packets are sent out and the client sends a report specifying how many bytes and packets it received. Using this mode it is possible to test for the maximum bandwidth that may be achieved across two computers.

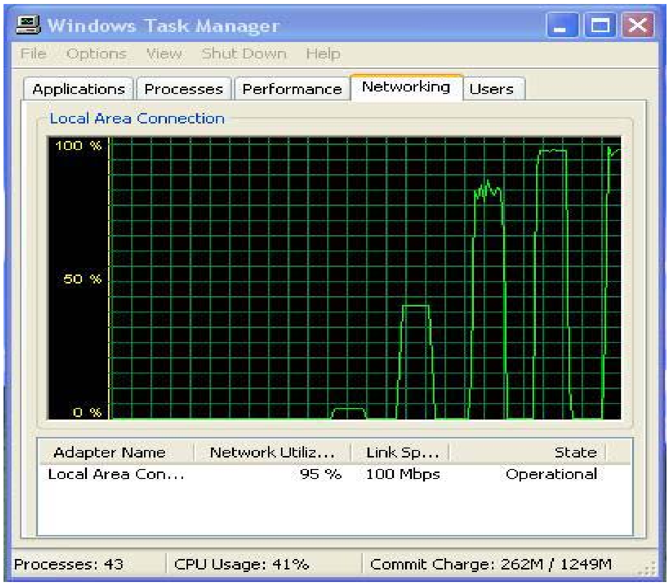


```
Current configured as Server
Press T to toggle mode
Press P to setup ping test
Press B to setup bandwidth test
```

```
Server Port available
Client Port available
Network Test initialized
Client successfully init
Server successfully init
Starting bandwidth test
Waiting for replies test
Client 192.168.3.75 P: 3523 Discover Reply
KBytes send this second 49000
KBytes send this second 50400
KBytes send this second 49000
KBytes send this second 50400
KBytes send this second 49000
Total Bytes sent 247800 Total packets 177
Client TotalBytes 247800, TotalPackets 177
Total test time is 5000 milleseconds
```

Figure 6-4 Bandwidth Test

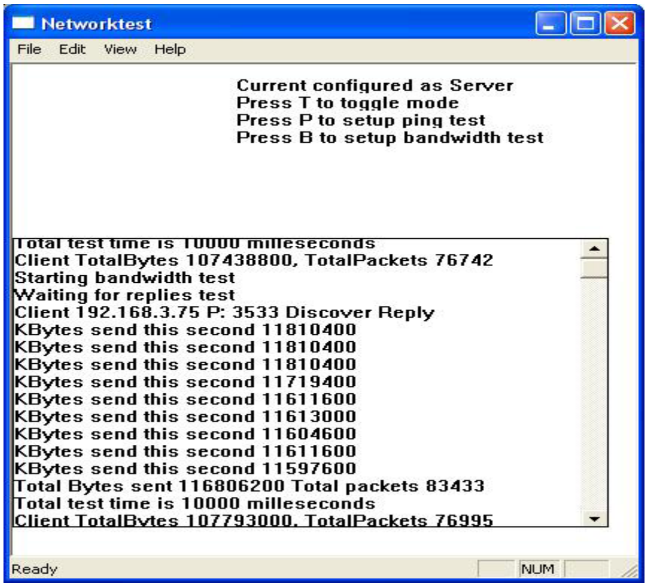
Figures 6-5 and 6-6 show a network test verifying that the maximum bandwidth of a wired 100Mbps connection is around 12 MBps. Note that when the network connection is stressed, the client will no longer receive every byte the server sent.


Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Local Area Connection
100 %
50 %
0 %

Adapter Name	Network Utiliz...	Link Sp...	State
Local Area Con...	95 %	100 Mbps	Operational

Processes: 43 | CPU Usage: 41% | Commit Charge: 262M / 1249M

Figure 6-5 Windows network activity



```
Current configured as Server
Press T to toggle mode
Press P to setup ping test
Press B to setup bandwidth test
```

```
Total test time is 10000 milleseconds
Client TotalBytes 107438800, TotalPackets 76742
Starting bandwidth test
Waiting for replies test
Client 192.168.3.75 P: 3533 Discover Reply
KBytes send this second 11810400
KBytes send this second 11810400
KBytes send this second 11810400
KBytes send this second 11719400
KBytes send this second 11611600
KBytes send this second 11613000
KBytes send this second 11604600
KBytes send this second 11611600
KBytes send this second 11597600
Total Bytes sent 116806200 Total packets 83433
Total test time is 10000 milleseconds
Client TotalBytes 107793000, TotalPackets 76995
```

Figure 6-6 Network test maximum bandwidth

6.2 Basic Network Testing

There are three possible types of networks between the server and client. One of them is a 100 Mbps wired connection (standard Ethernet). The other two are the wireless connections. The wireless network consists of a 100 Mbps wired connection to a Cisco aironet 1200 router connected wirelessly to a Linksys wireless USB adapter. Both the router and the adapter can run at multiple transmissions speeds including the two speeds that are used, 11 Mbps and 54 Mbps. These represent 802.11b and 802.11g network types. The speed can be changed by altering the transmission speed using the router's configuration page. There is no option of altering between mixed mode (both g and b) and "g" only mode but because of the high throughput it appears that it operates in only "g" mode.

Several test are preformed on the network to provide a base line for the experimentation later in this chapter. Explanations of the ping test and the bandwidth test follow.

6.2.1 Ping Test

The ping test records the ping time for the three kinds of networks. Two different ping test are used. One for sending 32 bytes forward and then 32 bytes back and the other for sending 32 bytes forward and 1400 bytes back which is more realistic for a resend request. There is a fair bit of variance in the ping results. Sometimes one ping time would be several times larger then the other. This is because the response time of the operating system can vary and there might also be other traffic on the network. The numbers should be taken as being approximate. Each test is 25 pings in length and the interval is 100ms. All tests are run with nothing else running on the computer at the same time, and with no other generated network traffic. The results are shown in Table 6-1.

Network type	Ping response time (32,32)	Ping response time (32,1400)
11 Mbps Wireless	1696 uS	3300 uS
54 Mbps Wireless	1345 uS	2119 uS
100 Mbps Wired	790 uS	1052 uS

Table 6-1 Ping test average results

6.2.2 Bandwidth Test

The bandwidth test checks the maximum available bandwidth on each of the three network types. The bytes per packet is set to 1400 and the test runs for 10 seconds. The speeds recorded in Table 6-2 are somewhat higher then the ones referenced by Atheros [Athe03a] in Table 6-3 as the maximum theoretical bandwidth, but overall the results are consistent. Different equipment vendors use different chipsets, which may have implemented the 802.11 standard, in particular the inter-frame spaces timing differently.

Network type	Maximum Bandwidth
11 Mbps Wireless	1034 Kbytes/second
54 Mbps Wireless	3984 Kbytes/second
100 Mbps Wired	11648 Kbytes/second

Table 6-2 Bandwidth test

Network type	Theoretical maximum Bandwidth
11 Mbps Wireless	887 Kbytes/second
54 Mbps Wireless	3813 Kbytes/second

Table 6-3 Theoretical maximum bandwidth

The next test checks to see what happens when the packet size is reduced from 1400 all the way down to 32. This is shown in Figures 6-7 and 6-8 Figure 6-8 is the same as Figure 6-7 except it does not contain the 100 Mbps to highlight the other two components. As the packet size is reduced, the throughput decreases. Note that the rather large drop in throughput for the 100Mbps link is due to CPU loading. At around the drop-off point of 200 bytes per packet, the CPU became 100% loaded and had difficultly handling the shear number of packets.

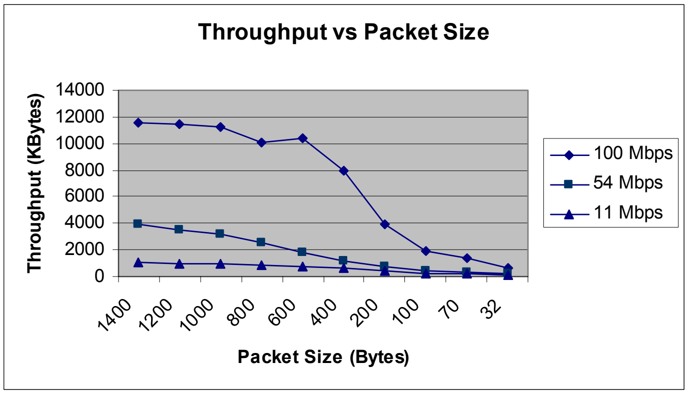


Figure 6-7 Throughput vs packet size 1

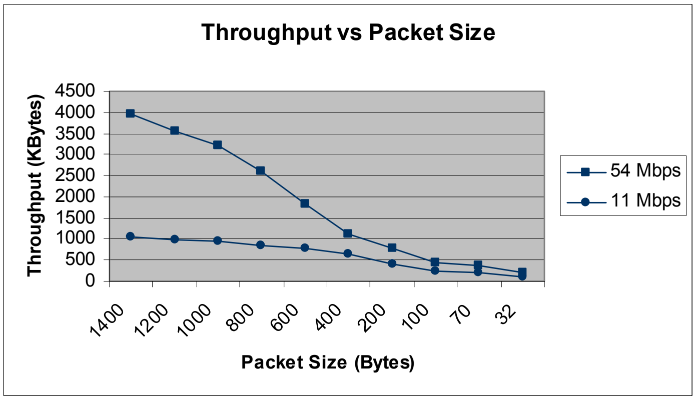


Figure 6-8 Throughput vs packet size 2

6.3 Media Application Installation Instructions.

6.3.1 Server Installation Instructions

In order for the server to work a capture card must be present on the computer. In this case the capture card is an ATI TV wonder pro. The proper encoders must be installed, so the Microsoft encoder codecs from Microsoft's web page should be downloaded and installed. Since the LAME audio encoder is used it has to be installed.

Before any graph file can loaded, the Send filter must be registered with the operating system. In command prompt, the command is "regsvr32 sendfilt.ax". A directory must exist called c:\profiles. In this directory, several configuration files are stored. Server.grf, the filtergraph file of the server is to be placed in this directory. Also before the server is used, the capture card must be conFigured to the proper settings, e.g which channel and which feed (s-video, composite) is to be used. This can be done in Graph Edit or using any utility just to verify that the video feed is coming in. The profile directory also must contain all of the encoder settings files, in this case enc80, enc85, enc90, enc95, enc80b, enc85b, enc90b, and enc95b which corresponds to the 8 preset encoder settings.

6.3.2 Client Installation Instructions

The client also requires the c:\profiles directory and the codecs to be installed. The two client filters nrecv.ax and Afilt.ax should be installed using regsvr32. The Client graph, Client.grf has to be present. One additional file, ftype.typ is required. This file is automatically generated by the server and specifies the codec formats that are sent so the nrecv filter knows how to connect up to the decoder.

6.4 Server and Client statistics

This section describes the various statistics generated by the server and the client.

6.4.1 Server Statistics

The server generates a statistics file containing the statistics gathered by the server over time. Below is an example of this file plus descriptions of each entry.

```
Server stats
Quality at: 80  Key frame rate at: total seconds: 2
TotalVideoBytes 5164418
TotalAudioBytes 938312
TotalIFrameBytes 668940
TotalPFrameBytes 4319038
TotalRBytes 4388
TotalIFrames 60
TotalPFrames 1533
TotalVBytes per second
1 20803
2 94383
3 102807
4 79541
…
```

- The second line contains information on the key frame rate and the VBR quality percentage.

- *TotalVideoBytes* is the total number of bytes sent out that were video including the header bytes required to send them.
- *TotalAudioBytes* is the total number of bytes sent out that were audio including the header bytes required to send them.
- *TotalIFrameBytes* is the total number of bytes that were video I Frames.
- *TotalPFrameBytes* is the total number of bytes that were video P Frames.
- *TotalRBytes* is the total number of bytes that were resend requests.
- *TotalIFrames* is the total number of I Frames used. Note that the codec might create I Frames on a non timed interval if the scene changes.
- *TotalPFrames* is the total number of P Frames used.
- *TotalVBytes* per second shows the number of video bytes generated for each second of the video stream. This is just to show the variance in the video stream.

6.4.2 Client Statistics

The client also generates a statistics file containing the statistics gathered by the client over time. The file is similar to the server's file except it contains more information. This is because it tracks if any video and audio frames that are missing when they should be played. Below is an example of this file plus descriptions of each entry.

```
Client stats
NOTAllowingresends packetlossrateat :10
TotalVideoBytes 2989314
TotalAudioBytes 957312
TotalIFrameBytes 600200
TotalPFrameBytes 2389114
TotalRBytes 0
TotalIFrames 475
TotalPFrames 2490
TotalAFrames 2493
```

```
TotalIDropped 0
TotalIBDropped 0
TotalPDropped 282
TotalPBDropped 485755
TotalADropped 0
TotalABDropped 0
TotalVMissing 276
TotalAMissing 0
TotalVBytes per second
1 38013
2 45931
3 57039
4 46640
5 39947
…
TotalAPFramesmissingpersecond
1 0
2 0
3 0
4 0
5 0
…
TotalVFrmesmissingpersecond
1 4
2 5
3 6
4 4
…
```

- The second line states whether or not the client is allowing resends as well as specifying the packet loss rate.
- *TotalVideoBytes* is the total number of bytes received that are video.
- *TotalAudioBytes* is the total number of bytes received that are audio.
- *TotalIFrameBytes* is the total number of video bytes that are I Frames.
- *TotalPFrameBytes* is the total number of video bytes that are P Frames.
- *TotalRBytes* is the total number of bytes that are from resend requests.
- *TotalIFrames* is the total number of I Frames received.
- *TotalPFrames* is the total number of P Frames received.
- *TotalAFrames* is the total number of Audio frames received.
- *TotalIDropped* is the total number of I Frames that are simulated to be dropped.
- *TotalIBDropped* is the total number of I Frames bytes that are simulated to be dropped.

- *TotalPDropped* is the total number of P Frames that are simulated to be dropped.
- *TotalPBDropped* is the total number of P Frame bytes that are simulated to be dropped.
- *TotalADrooped* is the total number of audio frames that are simulated to be dropped.
- *TotalABDropped* is the total number of audio frames bytes that are simulated to be dropped.
- *TotalVMissing* is the total number of video frames that are missing. Note that this number is lower then the TotalIPDropped because more then one packet for each frame can be dropped.
- *TotalAMissing* is the total number of audio frames that are missing.
- *TotalVBytes* per second is the average number of video bytes that are played each second.
- *TotalAFrames* missing per second is the average number of audio frames that are missing per second.
- *TotalVFrames* missing per second is the average number of video frames that are missing per second.

6.5 Basic Video Streaming Properties

The video streaming software is run for 60 seconds using the same 60 seconds of video to evaluate the different encoder rates. The encoder was changed to run at five different settings from 80% Variable Bit Rate (VBR) encoding quality to 100% VBR quality. Unfortunately, although Constant Bit Rate (CBR) encoding is quite useful to guarantee the bit rate over time, the encoder used had a limited range for the CBR modes, which made it unusable. Hence, no CBR rates are tested. Figure 6-9 shows the bit rate obtained for different quality settings. Notice how significantly the bit rate increases in certain time intervals. Also, as the quality approaches 100%, both the 100% and 95% use a significant amount of CPU cycles. At 100% VBR, the CPU is at 100% usage, which is bad. When the CPU is loaded to 100% usage, the encoder starts to miss frames and the overall quality goes down. Even at 95% VBR, there are occasional compression errors due to the CPU loading.

Figure 6-10 shows the bit rate for the 90% quality setting for 10 minutes. This setting uses around 50% CPU cycles on average and the video quality is quite good. This quality setting is used for further tests. Note that there is a significant amount of variance during the 10 minutes of video. There are many reasons for this such as some video frames are more detailed, and there is switching from the show to the commercials. Also, the faster the motion of the video, the higher the bit rate.

The variance in the bit rate, as shown in Figure 6-10, is quite high. This setting is unsuitable for bandwidth limited channels where some sort of peak bit rate guarantee is required. This is unfortunate because there is no other setting to run experiments on. The limited CBR settings available are unsuitable for testing. The encoder does have a peak bit rate version of VBR, but it requires two pass encoding which is unsuitable for live video. For our purposes the 90% VBR is fine and it is used in both of the two test scenarios.

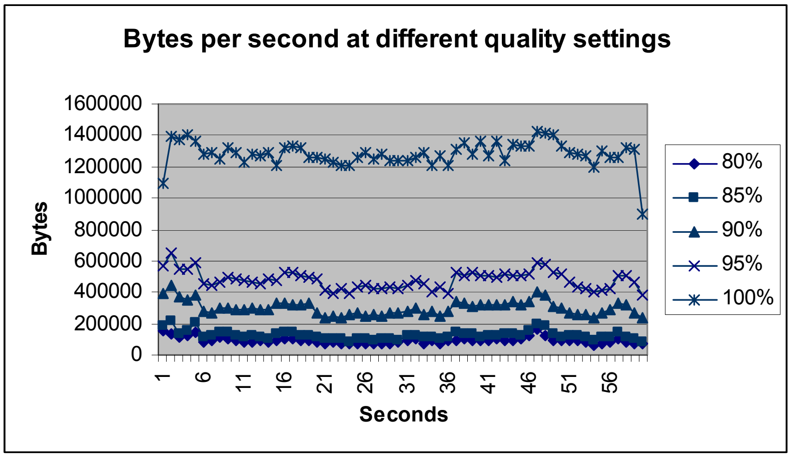


Figure 6-9 Video quality bit rates

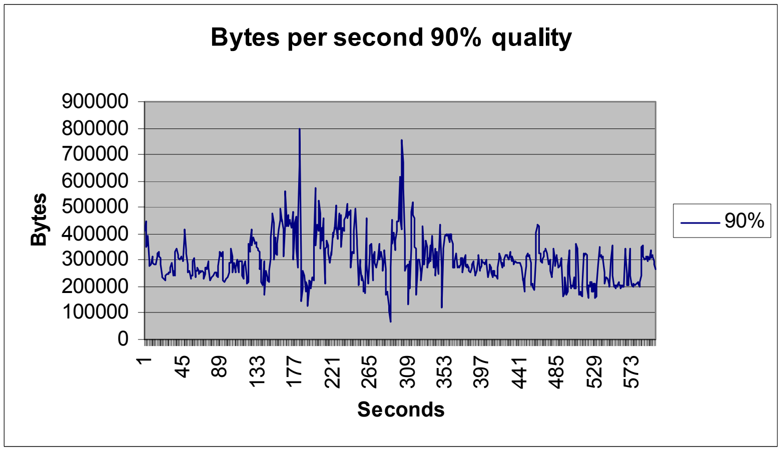


Figure 6-10 90% quality bit rate

6.6 Bandwidth Stress Test.

The first experimentation scenario addresses a very basic issue. What happens when there is insufficient bandwidth to transmit the video at certain points in time? This experiment uses the 90% quality VBR setting and simulated constant bit rate traffic generated by the network test utility. The ten minutes of video used for this experiment are the same that are uses that Figure 6-10. The level of traffic is set to 500 Kbytes per second. This value is chosen because the simulated traffic and the video added together will reach over 1 Mbyte per second a few times and this is higher than the maximum throughput for 802.11b. The buffers on the client end hold 5 frames, which means that throughput disruptions of more than approximately 150ms, cannot be compensated for by the resend request mechanism.

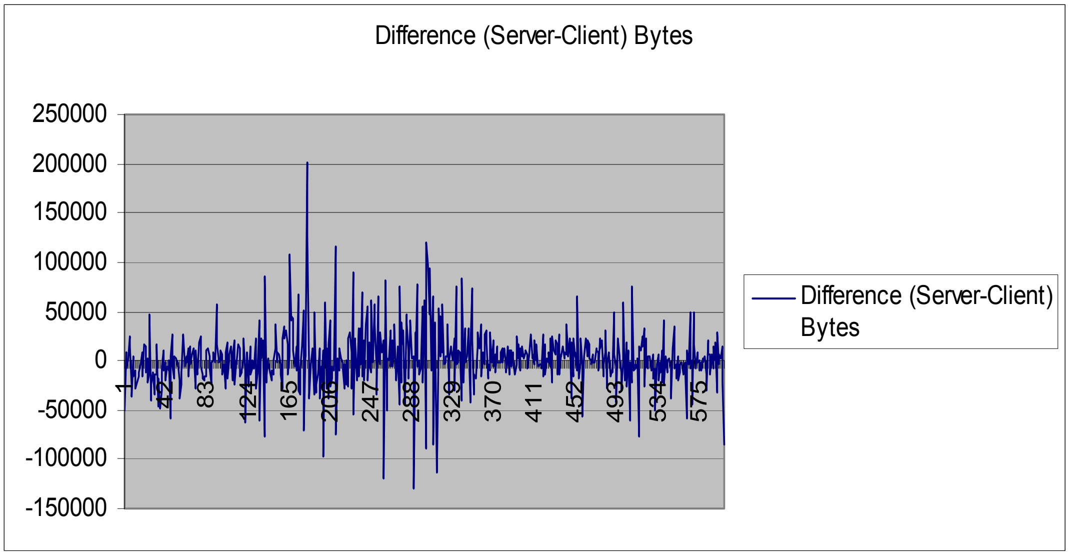


Figure 6-11 Difference in sending and receiving bytes

Figure 6-11 shows the difference between the number of bytes the server sends, and the number of bytes the client receives. The one problem is that time indexes of the server and the client are not aligned. What is evident however, is the differences at around 180 and 300 seconds. These correspond to the two peaks in Figure 6-10. This is further shown in Figure 6-12, which shows missing video frames in the same two peaks. Note that there are other instances where a few video frames went missing but those are probably due to the fact that it is impossible to guarantee the quality of a wireless connection over long periods of time.

In total, after 10 minutes, 181 Mbytes of video with 10 Mbytes of audio are generated and sent. The network test utility generated 297 Mbytes of traffic with about 289 Mbytes getting through, a difference of around 7.5 Mbytes not sent due to traffic congestion. Out of the video, 1.5 Mbytes is never received and this is after 2.6 Mbytes were taken up by resend requests. In all 248 video frames were missing and 181 audio frames missing.

The same experiment is run again, this time with much larger buffers. Instead of the 150ms buffers the size was increased to a very large 4 second buffers. This value was decided by looking at the peak bit rate of the video which never exceeds around 500 Kbytes over 3 seconds. Ideally, using a buffer this big should mean that all of the video frames will get to the client eventually and before their playing time. The results of this test is that no frames are missing during the 10 minutes. Of course 4 seconds of buffering is unrealistically large in a real time situation such as the amount of time that would be necessary for changing the channel in a TV application. But it does show that if the average throughput is lower then the maximum throughput, the peaks of the traffic could always be evened out with a large enough buffer.

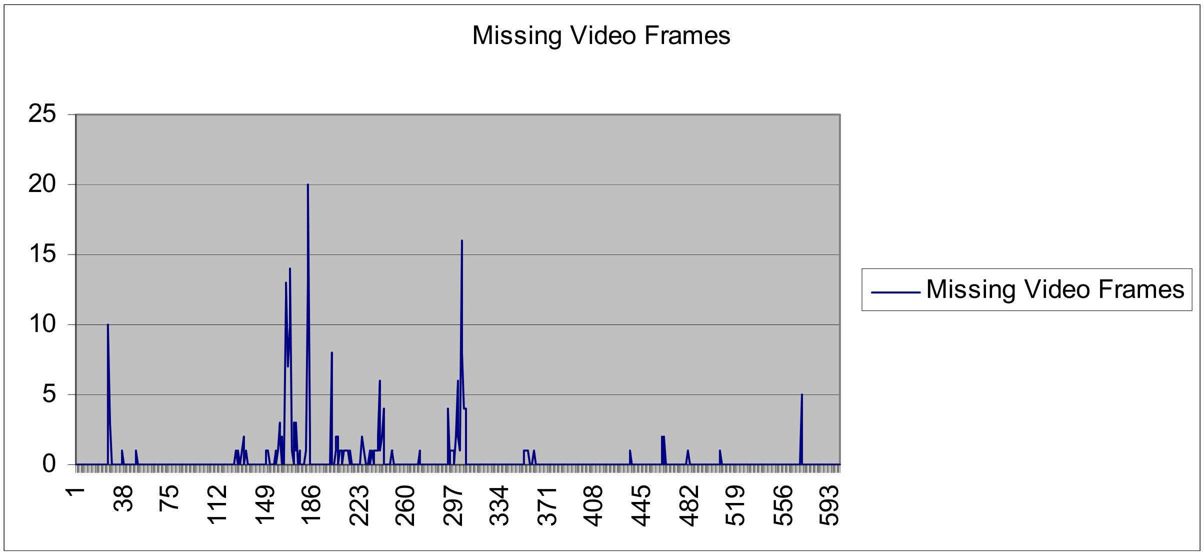


Figure 6-12 Missing video frames

6.7 Error Rate Contrast with 802.11b and 802.11g

The next scenario that is being tested is as follows. The media streaming is running at 90% VBR quality over an 801.11b and 802.11g networks, that otherwise has no traffic on them. The buffering on the client is configured to be at 2 frames including the internal buffering that corresponds to around 150ms. This time is selected because a small latency like this is useful in interactive applications or even mundane ones, such as changing the channel on a TV. The client is configured to have various packet loss rates. The question is, how does the increase in the simulated packet rate decrease the video quality as measured by the number of video frames unavailable to be played?

6.7.1 801.11b with 1% Packet Loss Rate.

Out of a total of 137,892 packets a total of 1,372 were dropped (which is around 1%). The distribution is shown in Figure 6-13 which verifies that the drops are indeed random. The number of video frames missing is not random though. There are two peaks at 182 and 302 seconds, which correspond to the peaks in traffic as shown by the previous Figure 6-10.

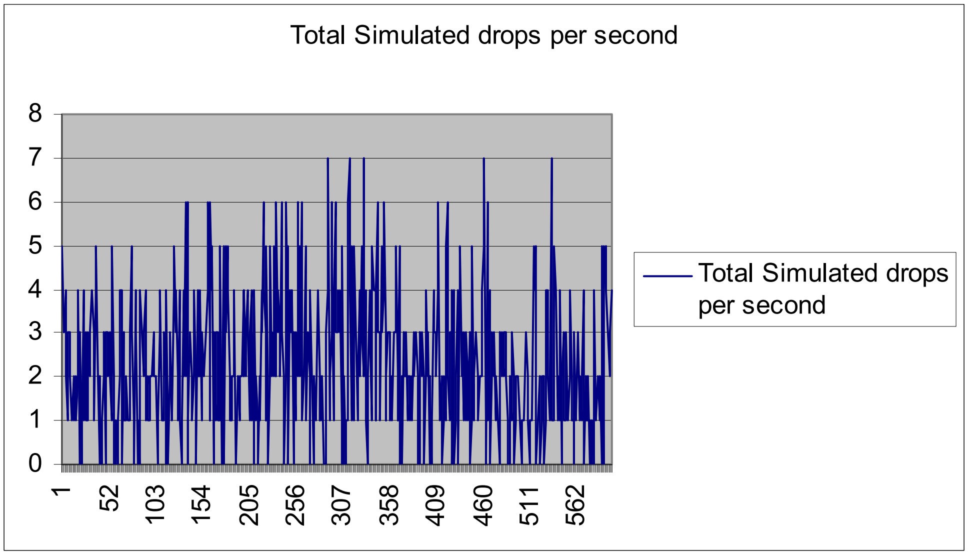


Figure 6-13 Total simulated drops per second (11 Mbps 1%)

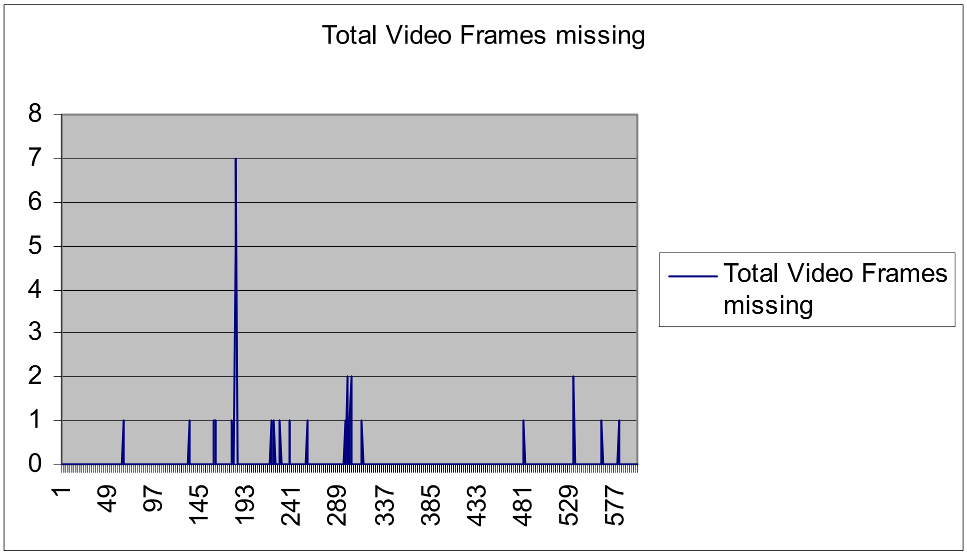


Figure 6-14 Total video frames missing (11 Mbps 1%)

6.7.2 802.11 with 5% Packet Loss Rate

The 5% packet loss rate test is far more interesting. At this higher rate, far more video frames were missing under the normal traffic load. But something else happened. In the two peaks (182 and 302 seconds), there was not enough bandwidth for both the video stream and the resend requests. When this happened so much bandwidth is taken up by the resend requests, that not enough is available for the transmission of traffic, that would normally be sent correctly (i.e. not dropped by the simulated packet drop). This is shown in Figure 6-15 where there is a small frame missing rate normally but that value sky rockets during the two peaks. With so many frames missing during the two peaks, the video becomes unwatchable and mostly just freezes. In addition, due to the loss of available throughput, the audio packets, which are not subject to simulated drops start to be missing. This is shown in Figure 6-16.

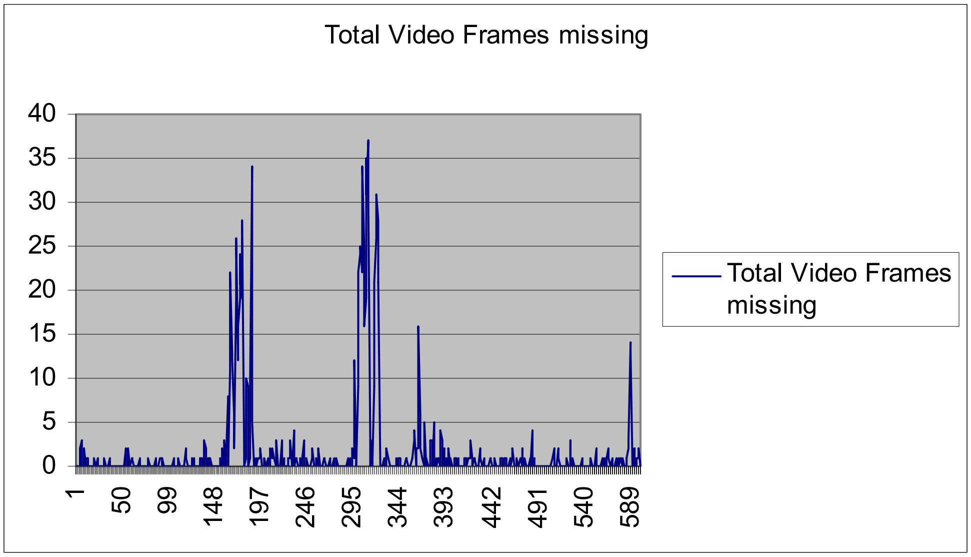


Figure 6-15 Total video frames missing (11 Mbps 5%)

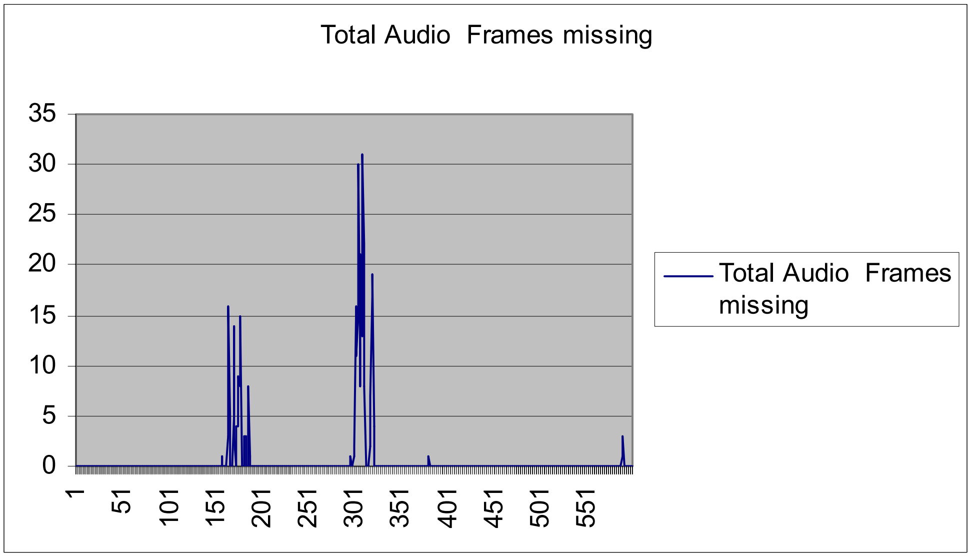


Figure 6-16 Total audio frames missing (11 Mbps 5%)

6.7.3 802.11g with Varied Packet Drop Rate

The same experiments are now run with the 802.11g network. This network proves to be much more resilient than the 802.11b network. Because of this, additional tests are run while increasing the packet drop rate to 20%.

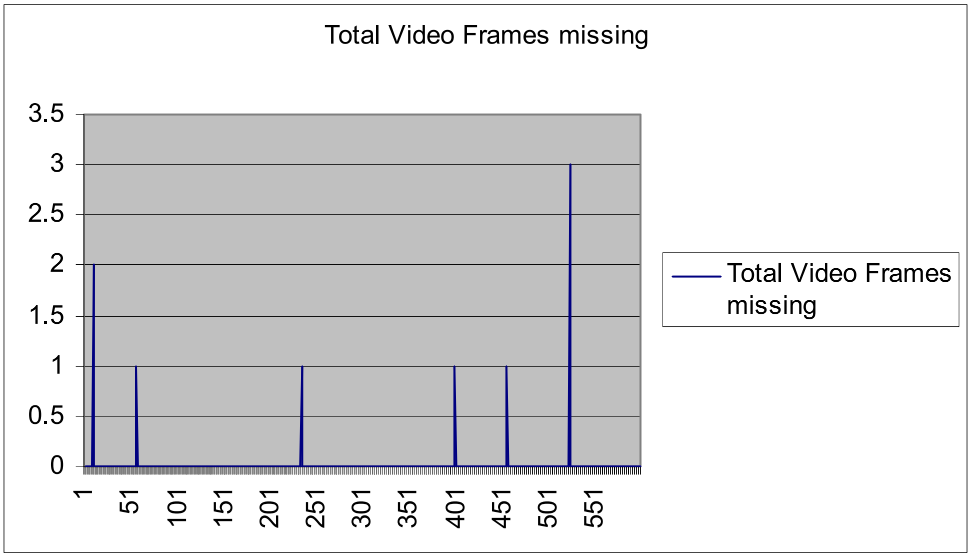


Figure 6-17 Total video frames missing (54 Mbps 1%)

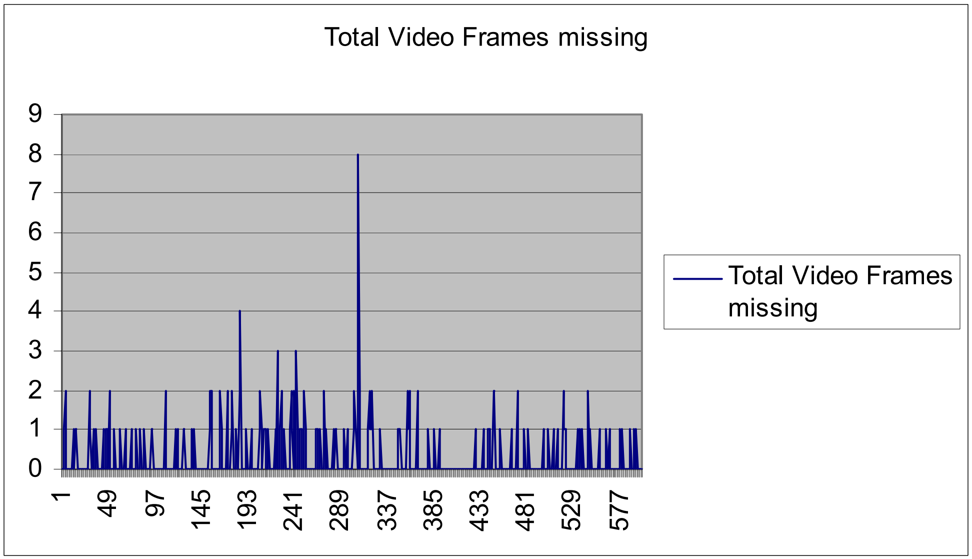


Figure 6-18 Total video frames missing (54 Mbps, 5%)

Far fewer frames are dropped in Figure 6-17 and there does not appear to be a correlation with increasing amounts of traffic. In Figure 6-8, when the packet drop rate is 5%, there is a slightly higher correlation with the traffic, as seen by two peaks. Further test were run at 10% and 20%, with 20% being the more interesting as shown in Table 6-4. Figure 6-19 again shows the traffic at 90% VBR for 10 minutes. Figure 6-20 shows the simulated packet drops over time and Figure 6-21 shows the frames missing over time. These graphs look similar except for the scale. With such a high loss rate at 20%, the density of the random drops is enough to for the missing frame rate in a given second to directly correspond to the number of bytes sent that second. Note that very few, if any, audio frames were ever dropped, so any missing frames is entirely due to dropping the same packet twice thereby making it unavailable when it is required.

	(54 Mbps 1%)	(54 Mbps 5%)	(54 Mbps 10%)	(54 Mbps 20%)
Total Video Traffic	180122982	183884042	186920318	186512921
Resend Traffic	1772250	9565591	21156994	44616837
Percentage	.984%	5.2%	11.3%	24%
Total missing frames	9	172	988	3954

Table 6-4 Loss rate VS Missing frames

Table 6-4, shows the number of bytes used to resend the packets given a simulated drop rate. Note, as the drop rate becomes higher, a larger difference occurs in comparing the simulated drop rate with the quantity of bytes used to resend packets. This further confirms that the higher the percentage of dropped packets, the greater the likelihood of the same packets being dropped more than once, and being attempted to be resent more than once.

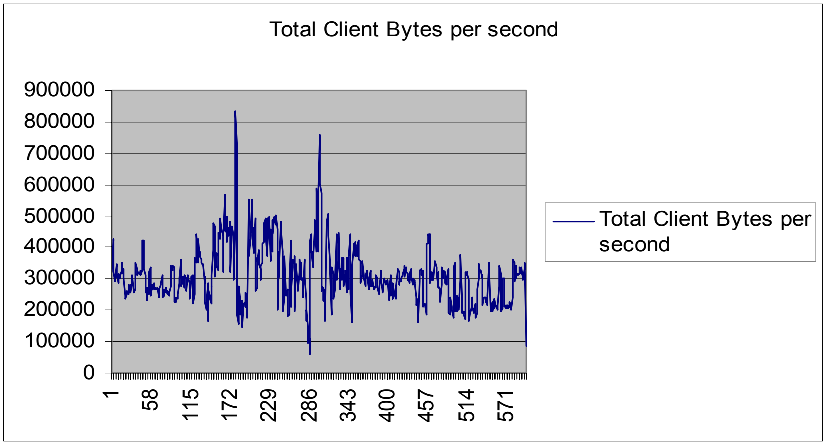


Figure 6-19 Total Client Bytes per second (54 Mbps 20%)

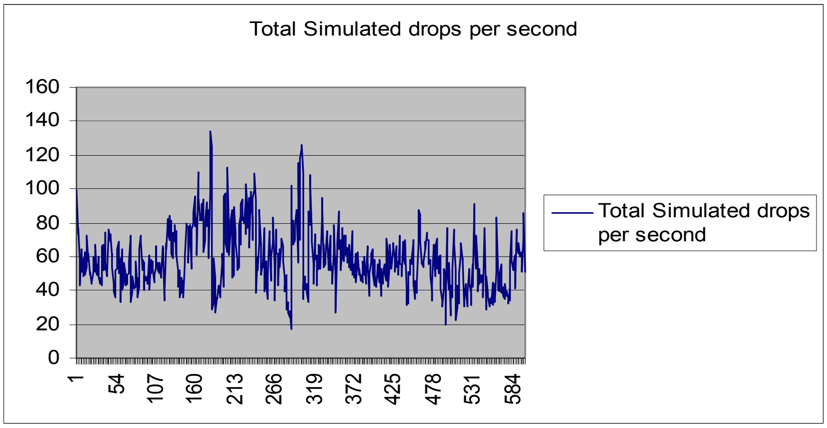


Figure 6-20 Total Simulated drops per second (54 Mbps, 20%)

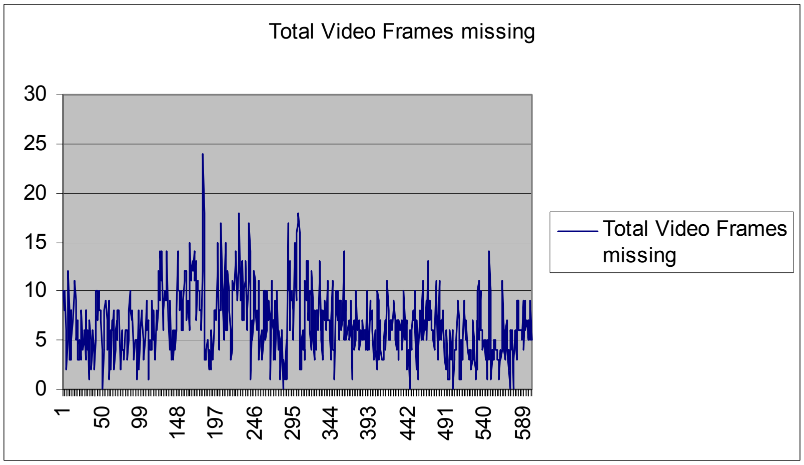


Figure 6-21 Total video frames missing (54 Mbps, 20%)

6.7.4 Simulated Packet Drop Conclusions

The two different networks, 802.11b and 802.11g, show two different results in terms of packet loss. The 802.11b network is more bandwidth limited so the testing readily shows the effects of packet loss when the throughput limit is being reached. When this happens, there is a cascade failure. The resend request mechanism is doing more damage than good because it increases the used available throughput considerably.

802.11g has a large excess of throughput when only the one media signal is being transmitted. Therefore, it does not show this cascade effect because the throughput limit is never reached. Because the missing frame rate was low, it was possible to increase the simulated drop rate to see its effects. As the simulated drop rate increases, both the dropped packet and missing frame rates pretty much approach the level of the traffic. This makes sense, as the larger the traffic, the greater the quantity of packets being generated, therefore, the greater the quantity of packets being dropped. Any frames that are missing are entirely due to the fact that the resend attempts of a given packet are dropped, and not because of any throughput problems.

Chapter 7 Conclusions

7.1 Summary

This thesis provides a thorough background and in depth description of wireless media streaming. This section highlights some of the more interesting points from different sections of this thesis.

The component based architecture used in the wireless simulator proved very useful. It decreased the overall time required to develop the simulator, particularly in terms of less time spent debugging. This architecture also allowed for the easy expansion of the simulator to use multiple channels. The architecture made it possible to integrate an API. To program the simulation only requires expanding a single inherited class without requiring any knowledge of how the rest of the simulator works. The simulator API is simple and resembles popular OS such as Windows or Linux.

The experimentation using the wireless simulator showed several things. The first is that fundamentally, wireless in an application programmer's perspective can be modeled very similarly to wired networks with higher latencies and lower throughput. Even with a higher BER, the RTS-CTS-Data-ACK mechanism guarantees that the data is eventually sent. Adding an access point decreases the throughput by half, and increases the latencies by more than two, but does not add any caching effects. When streaming video, it is important that when frames are fragmented into packets that they are all sent. If there is not enough capacity, whole frames should be dropped by the server, until it is guaranteed that the entire frame consisting of several packets can be sent. When using multiple channels, it is observed that it effectively increases the throughput accordingly with some care being needed with the channel selection algorithm. If some other knowledge such as channel BER is available, that would likely be useful to incorporate in the algorithm.

The chapter on DirectShow filters is useful for anyone who wants to learn the basics involved in writing a filter. The chapter is a condensed version of what is available in the DirectX SDK. A good way to learn how to write a filter is to read the chapter and then try to get a memory copy filter working.

The discussion of how the media streaming software is developed should also prove useful in developing one's own software. Even if one does not use DirectShow, the architecture discussion should be useful. In particular, the transport component architecture is designed well. On numerous occasions during the development, different items were added to the packet headers in the header class, without having to make any changes anywhere else. It is also possible to reuse the code for different applications.

The description of the resend request mechanism description and synchronization mechanisms should prove useful. Although the need for these mechanism are discussed in [Perk03], their implementation is not.

Testing the wireless networks showed that for 802.11b it is possible to achieve around 1 MBytes per second in terms of throughput. For 802.11g it is possible to achieve around 4 MBytes per second. In both cases, until the throughput of the channel is used up, all of the data is sent reliably.

The media streaming experimentation chapter has some interesting points. The first is a graph of the VBR bit rate stream over time. The peak rates of VBR make it unsuitable if the channel requires guaranteed bit rates. However, the bit rate leads to some interesting experimentation.

When there is not enough throughput available for all of the traffic, and the buffering of the media stream is low, frames will be lost. However, if the buffer is set large enough to compensate for the peak bit rates, it is possible not to lose any frames.

Generally speaking, after observing many hours of video, it is far preferable to view video at a decreased bit rate rather than lose video frames. This is corroborated by [LMZ02]. The process of lowering the bit rate is nearly seamless. However, the far more interesting issue of knowing when to do this automatically is not explored in this thesis.

The packet loss tests are interesting too. They show that, for the most part, the packet resend mechanism works. One unexpected result was that when the resend request took up enough throughput, it caused the media streams to fail for a few seconds. This is very noticeable visually. Also it is shown that as the packet drop rate increases, the number of missing frames increases at a greater than linear rate. This can be explained because when the first packet drop will not cause the frame to be missing, but the second one will.

7.2 Recommendations for Future Work

1. Resend request mechanism: Having a smarter resend request mechanism could be useful. Measuring the exact time it would take to respond to a resend request could be useful. Also is could be useful to avoid the cascade failure when the resend request takes up enough bandwidth to cause a few seconds of media to be missing.

2. Fully componentize the wireless simulator: The core of the wireless simulator can be made into Dynamic Link Libraries and made into more of an SDK format where it would be very easy to distribute to other people.
3. Adding routing to the wireless simulator: The wireless simulator has the capacity to simulate the Internet. Since it relies on packets it is relatively easy to add Internet routing to the simulation.
4. Investigate throughput issues on a home network: Generally speaking, if there is insufficient throughput, the streaming just will not work. Since streaming is in real time and probably a higher priority then other home networking traffic, it might be prioritized at the MAC level. Experimentation can be done by using the simulator to investigate the new 802.11e standard, which includes different priorities of packets, each priority class having its own queue and back off timer. The simulator has the capacity to be easily expanded to having multiple queues and back off timers on a single network device.

References

[AhDa96] J. Ahn and P. B. Danzig, Speedup vs. simulation granularity *IEEE/ACM Transactions on Networking*, vol. 4, no. 5, pp. 743--757, Oct. 1996.

[Athe03a] 802.11 Wireless LAN Performance. 2003 Atheros Communications, Inc.

[Athe03b] Methodology for Testing Wireless LAN Performance with Chariot. 2003 Atheros Communications, Inc.

[Bren97] P. Brenner. A Technical Tutorial on the IEEE 802.11 Protocol. Breezecom Wireless communications 1997

[COM05] Component Object Model http://www.microsoft.com/com/default.mspx

[FPJF03] G. Flores-Lucio, M. Paredes-Ferrare, E. Jammeh, M. Fleury, and M. Reed. OPNET-Modeler and NS-2: Comparing the Accuracy of Network Simulators for Packet-Level Analysis using a Network Testbed. *3rd WEAS Int. Conf. on Simulation, Modelling and Optimization (ICOSMO 2003)*, Crete, vol. 2, pp. 700-707, 2003

[HBEI01] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. chan Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of detail in wireless network simulation. *In Proceedings of the SCS Multiconference on Distributed Simulation*, pages 3-11, January 2001.

[HuKH04] A. Hussain, A. Kapoor and J. Heidemann, The Effect of Detail on Ethernet Simulation*, PADS-04*, Kufstein, Austria, 16-19 May 2004.

[IEEE99] ANSI/IEEE Std 802.11 Wireless LAN Medium Access Control (MAC) and Physical Layer Specification, IEEE 1999.

[JaEl98] S. Jacobs and A. Eleftheriadis. Streaming Video using Dynamic Rate Shaping and TCP Congestion Control. *Journal of Visual Communication and Image Representation, Special Issue on Image Technology for WWW Applications*, 211-222, September 1998.

[JSIM04] J-Sim network simulation tool. www.j-sim.org

[LAME05] Lame MP3 encoder http://lame.sourceforge.net/

[LiZa04] H. Liu and M. El Zarki. On the adaptive delay and synchronization control for video conferencing over the Internet. *ICN'04*, France

[LMMZ02] Xiaoxiang Lu, Ramon Orlando Morando, Magda El Zarki, "Understanding Video Quality and its
use in Feedback Control," *Packet Video 2002*, Pittsburgh, PA, USA 2002

[Micr02] DirectX 9.0 SDK 2002 Microsoft Corporation.

[Micr04] Microsoft Research Conference XP http://www.conferencexp.com

[Micr05] Microsoft Media Home
http://www.microsoft.com/windows/windowsmedia/default.aspx

[Nede01] P. Nedeltchew. Wireless Local Area Networks and the 802.11 Standard. March 31 2001.

[Nico02] D. Nicol, Comparison of Network Simulators Revisited, Dartmouth College May 20,2002

[NS204] NS2 network simulation tool. http://www.isi.edu/nsnam/ns

[PeHH98] C. Perkins, O. Hodson, V. Hardman. A Survey of Packet-Loss Recovery Techniques for Streaming Audio. *IEEE Network*, pp.40, September 1998.

[Perk03] C. Perkins. RTP Audio and Video for the Internet. Addison-Wesley 2003, 414 pages

[Pesc03] M. D. Pesce, Programming Microsoft Directshow for Digital video and Television. Microsoft Press 2003, 413 pages

[Tall98] Talluri, Raj, "Error-Resilient Video Coding in the ISO MPEG-4 Standard*," IEEE Communication Magazine*, June 1998.

[Tyan02] H. Tyan. Design Realization and Evluation of a Component-Based Compositional Software architecture for Network Simulation. A Doctorate Thesis, Ohio State University 2002.

[ViLa04] Video Lan, Multioperating system video streaming software.
http://www.videolan.org/