24.765 802.11 Simulated Application Layer experimentation

Jay Kraut

Abstract:

This paper examines results for video streaming experiments over a wireless network. Building on an existing simulator architecture an API is created to allow for simulation of applications. This simulator is then used to simulate the real conditions that occurred in the experimentation. The QoS seen in the real experimentation is attempted to be duplicated, and then explained by the simulator results.

Keywords:

CSMA/CA, 802.11b, video streaming, network simulators.

1. Introduction

There is an increasing amount of convergence in the home entertainments 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 several manufactures such as Sony and Creative that are manufacturing devices that 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 the gradual elimination of wired networks in favor of wireless ones. Wireless networks are slower but have advantages. Wireless networks eliminate much of the clutter of connecting an increasing amount of devices with wires. Also wireless networks give added mobility, which is practical for devices such as laptops.

The streaming of video media to one of many wireless displays is becoming more common. If bandwidth is unlimited then reliable streaming video is relatively easy to implement. However, as the bit rate increases with higher fidelity video formats (HDTV) maintaining a higher QoS (quality of service) becomes difficult. This report looks at issues with streaming video media over a wireless network.

Two experiments were run to assess the performance of live streaming of video. The server encodes a MPEG-4 stream using DVD quality settings and streams it to a display that uses windows media player. In one test setup, (figure 1) both the server and the display are both connected wirelessly through an access point. In the other test setup, (figure 2) the sever is connected to the access point through a wired network, while the media display is connected wirelessly. The wireless connection uses the 802.11b standard.

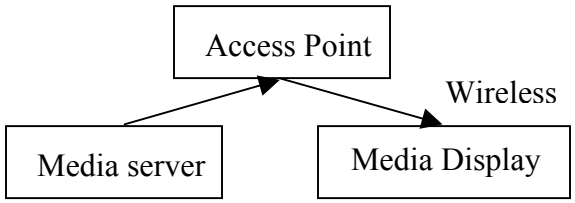

Figure 1 Test setup 1

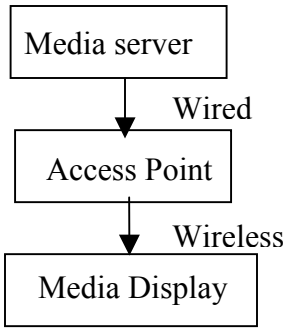

Figure 2 Test setup 2

Using test setup 2 the video stream is broadcasted correctly. Using test setup 1 the video stream does not work correctly. The audio comes through fine but the video is choppy. The video plays for around 5 seconds and then stops working after 5 seconds constantly repeating this on-off pattern. From

assignment #2 it is found that there is twice as much bandwidth using a wired rather then using a wireless server. This is because when the server is wireless, the server must send the information first to the access point and then the access point sends the information to the display. It was also found that other then halving the there is very little other inefficiency. One thing to note is that the wait time does goes up more then twice as much.

Using a simulator this report investigates why the effect of the on-off choppy video occurred in test setup 2. The simulator architecture is also discussed.

2. Simulator architecture

The internals of the simulator and the verification of its performance is discussed in assignment #2 and won't be repeated. However, the application programming interface (API) is discussed in this section.

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 detailed below should be very familiar as they emulate operating system calls.

2.1 Process Overrides

An application should implement the following functionality to operate 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

2.2 Event Overrides

These following functions are ones that should be overridden 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 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 the data so it can be automatically averaged and written out when a program is done executing.

2.3 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 cards message count are automatically stored and can be

written out. However in order to store any statistics that are generated by an application the following functions are used

*int GetCustomStats(Process *pProcess, char *pName)*
This function gets a 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 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 return 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

2.5 Timers

NGetTicks()
This function returns the current time in ms.

NGetUTicks()
This function returns the current time in us.

2.6 Socket Functionality

These function calls are made to be similar to other operating systems 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 create a socket and returns its ID.

void SetSockOpt(int Socket, int Option)
This function sets up options of 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.

2.7 Network setup functions

The network class only requires two functions to be overridden

void SetupNetwork()
This function setup up the 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 determine exactly which statistics to save.

The network has the following options that should be set and 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. E.g 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 is to increase traffic as *m_nSaveStats* is increasing so networking performance testing can be batched together.

3. Application programming

Using the API the video stream program is created as follows. The server is very simple. It simply sends out simulated data (that is not generated). The following code shows its primary functions

```
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 run function is called every 1 ms by the network. The first thing that is checked is to see if a new frame is ready to send. If it is, then is there a check to see if the previous frame has been fully sent. If it has, it sends out the next frame. If it has not it skips a frame. The server includes a timestamp of the frame simply to calculate the wait time and the frame number for placement in the receive buffer.

The client is a bit more complex because it is the one that has to generate most of the statistics. The application first uses the *OnMessageReceived* function to receive any data. It has a member class that acts as the video buffer. The class *m_VideoBuffer* parses the header of the data to gets its frame number. It then places the data into the proper place in the buffer.

```
void VideoClientProcess::OnMessageReceived(int SourceIP, int Port)
{
	if (Port == VIDEOCONTROLPORT)
	{

	}
	else if (Port == VIDEODATAPORT)
	{
		m_pOS->Receive(m_DataSocket,m_ReceiveBuffer,MAXMESSAGESIZE);
		m_VideoBuffer.AddData(m_ReceiveBuffer);
	}

}
```

The run function performs most of the grunt work.

```
void VideoClientProcess::Run()
{
	if (m_StreamingTime <= NGetTicks())
	{
		m_CurrBufferSize = m_VideoBuffer.GetNumFrames();
//		ASSERT(m_CurrBufferSize < 40);
		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 function runs on the rate in terms of frames per second as the servers. It first checks if it is time to run. If it is time, it then checks how many frames are available in the buffer. There is some hysteresis on this check. If it is already displaying frames a smaller number of frames are required. However if it has stopped because the buffer is too small it waits for the number of frames to go to a higher number before starting to play again. Even if there are sufficient frames in the buffer it is not guaranteed that the frame required is available. Because of congestion the current frame may not have been sent but the next one has. Depending if the data is valid the frame is played or not and various statistics are generated.

3.1 Caveats and other Notes

All of the traffic is UDP however the operating system supports fragmentation. This was done for convenience because most applications fragment UDP data because of its limited packet size. TCP is not going to be implemented because of the fact that the simulator is required to works only under LANS without any support for routing. This means that the latencies due to routing and packet losses due to routing are non-existent so TCP reliability is not required. The only two things that TCP implements that are potentially required are fragmentation, which is implemented in the simulator and rate control, which is implemented in the application level.

One thing that the conventional OS implements that is not implemented in the simulator is using call back function for event handling. When posting events the simulator posts the events to the *OnReceived* or *OnSent* function calls. This forces the user to check the Source IP and Port number to determine which socket the event relates to. This is accomplished through "if statements". Usually this can be accomplished through callback functions as they would replace the if statements, however because of the way the class hierarchy is setup it would be difficult and time consuming to implement. As of now it is not implemented.

4. Experimentation

The goal of the experimentation is to determine why when streaming video over a congested network the video stream fades in and out every 5 second or so. In order to do this a network is setup with one Server and one Client with an access point. The data is streaming at various rates. The frame size of the data is fixed at 20000 bytes per frame but the streaming rate is variable and is ran at 45ms to 30ms (22.2 – 33.3 fps). The original hypothesis that is tested is that because of the fact that the access point acts as a buffer the extra buffering causes the choppiness. This hypothesis also assumes that the video server performs no rate control and simply sends out frames blindly. The reason for this assumption is even though the protocol is undocumented by Microsoft, the setup used makes use of the free version of Microsoft software. Microsoft's advertises a premium version that uses RTP (real time protocol) so it is assumed that the free version doesn't have much in terms of QoS controls. If this hypothesis is true then the buffer will fluctuate between playing and not playing at an interval of 5 seconds at a certain point in traffic. The following statistics to verify the hypothesis are gathered.

Buffer size of the Client. There are two buffer sizes that are gathered. One of them is the size of the buffer in terms of the number of frames between the frame that is about to be played and the last frame in the buffer. The other is the size of the buffer in terms of frames that are playable over the same range as before. This is an indication as to how many frames are missing in the buffer.

Skipped frames. This is how many frames are skipped because the buffer is too small to play.
Wait time. This is calculated as the time between when the server creates the frame and the time it is fully received by the client

Run Length and number of runs. This is the length of frames being run continuously or being missed continuously. For example when 10 frames are played then at the 11 one it is missing a run of 10 has occurred. This indicates the quality of the stream. Note that run length was capped off at 50 at which it starts counting again.

Played Frames and Percentage of frames that are playable. This tracks how many frames are played over the same period over different network configurations.

To verify the original hypothesis the simulator would be tested under various buffer sizes to see if the 5-second choppiness could be generated. Figure 3-12 show the buffer sizes and run length for the different frame rates. The 45ms is just before congestion and the 30ms is at heavy congestion. When congestion rises the following is observed. The higher the congestion the lower the total buffer size to playable buffer size. This is further verified by the run length. At heavy congestions very few full frames are transmitted correctly. At 30ms the buffer becomes so far behind it is flushed. This can be observed as the buffer size drops to 0 for a while.

The reason for this happening is that in heavy congestion the buffers are nearly always full. So when trying to send a message only some of the fragments can be entered into a buffer so most of the messages are not correctly sent. This is verified by figure 13 and 14 at which as the congestion gets heavier the number of packets sent correctly gets lower. To mitigate the fragmentation the data is sent only if it was possible to send it unfragmented. The end results of this is under congestion every nth frame would be sent similar to run length as shown by Figure 19,20. However the overall performance of not fragmenting the frames is shown by Figure 16-18 to be higher.

However after the first run of variable congestions (45-30ms) the it seem that it is unlikely that the original hypothesis is correct. Under heavy congestion there is no sign of having a buffer under run for 5 seconds. Its seems that unless the access point implements some sort of buffer flush, the 5 second under run can not be caused by any buffering in any of the lower levers of 802.11.

At this point a packet sniffer is used to see what the network traffic looks like. The first thing that is noticed is that the traffic is TCP not UDP. But even with TCP acks, windowing and rate control it wouldn't account for 5 second under runs. The new hypothesis is that the buffer under runs is caused at the application level.

There must be a reason that the application would prefer to send video data at 5 second intervals rather then send out every nth frame. An examination of video compression can explain this. MPEG video compression sends 3 kinds of frames. I-Frames (intraframes) stores the full frame as a compressed picture. B-frames (bi-directional) and P-frames (predictive) are based on motion prediction from the I-Frames. It seems reasonable that in the application level having only every second frame being sent would be pointless because each frame requires the one before it (except I Frames). In order to display any sort of video a group of frames would have to be sent over correctly together. If only every second frame is sent it is likely that nothing would be displayed.

So that explains why there are 5 second under runs. The application takes a look at the congestion (can be done under blocking UDP calls too). At some point it time the application realizes that for the next few seconds it will be unable to send the data quickly enough. Because it has some logic that tells it that it must send frames over in chunks it simply advances its own clock and sends data that is several seconds ahead. Even though a gap will happen at the client some data will get played which is better then nothing.

5. Future Work

It is not possible to fully verify the new hypothesis without doing actual testing of video streaming. Later on any protocol used in the simulator might be implement in the video streaming sever to verify that the hypothesis is correct.

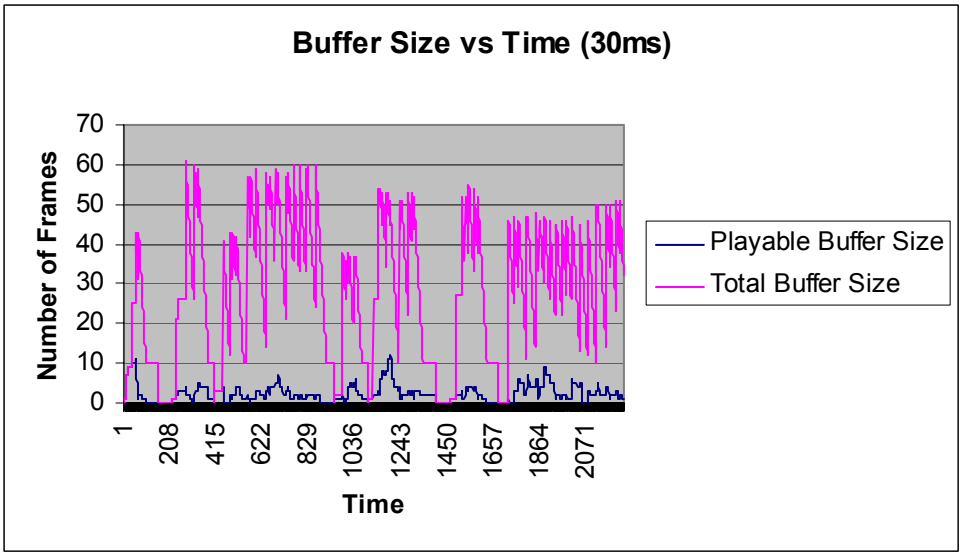

Figure 3.

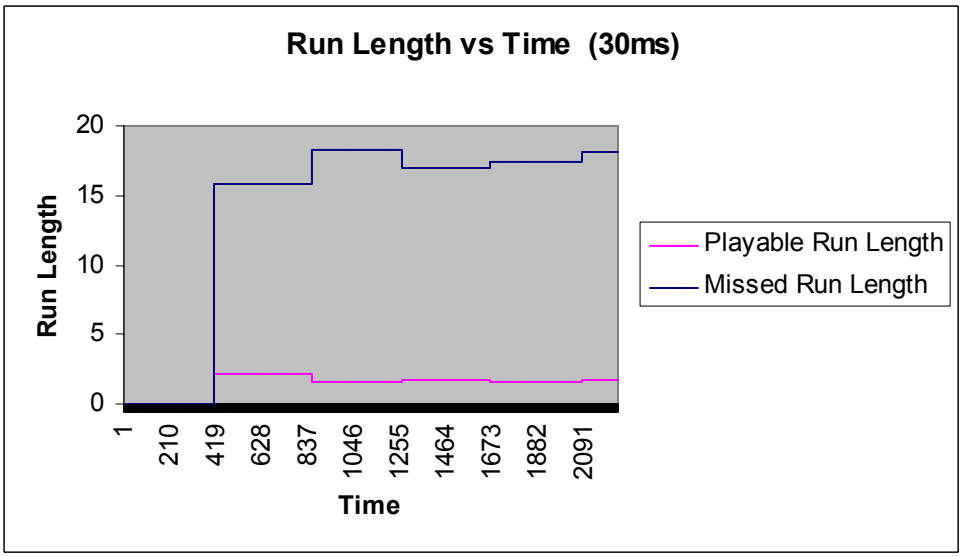

Figure 4.

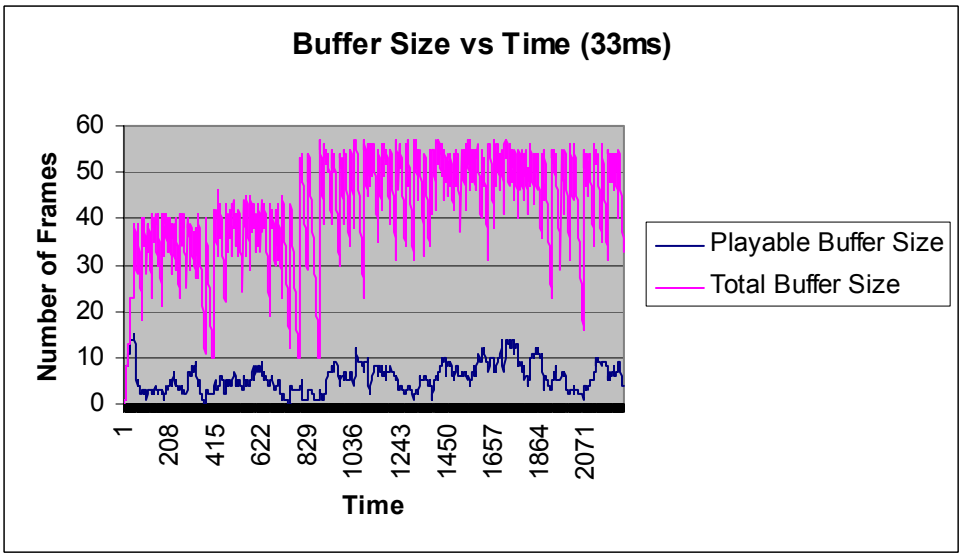

Figure 5.

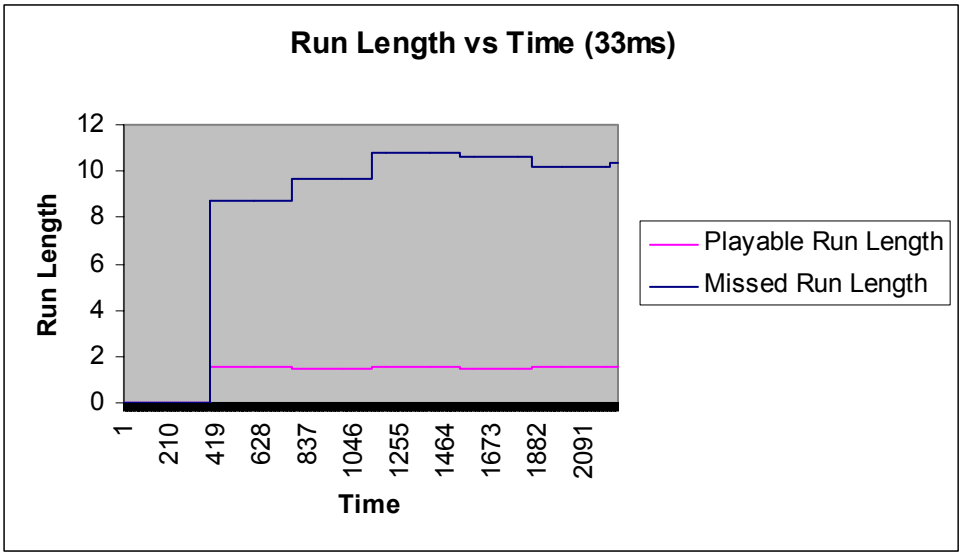

Figure 6.

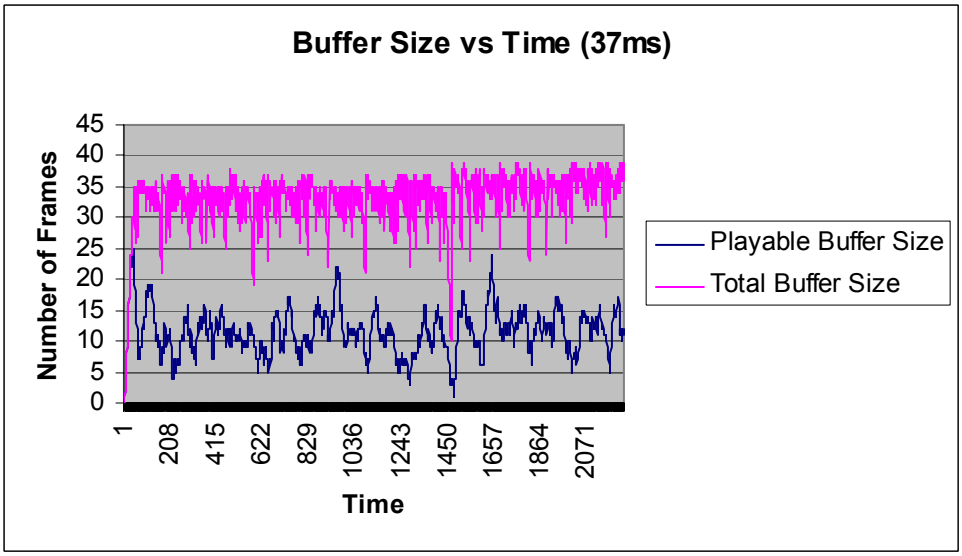

Figure 7.

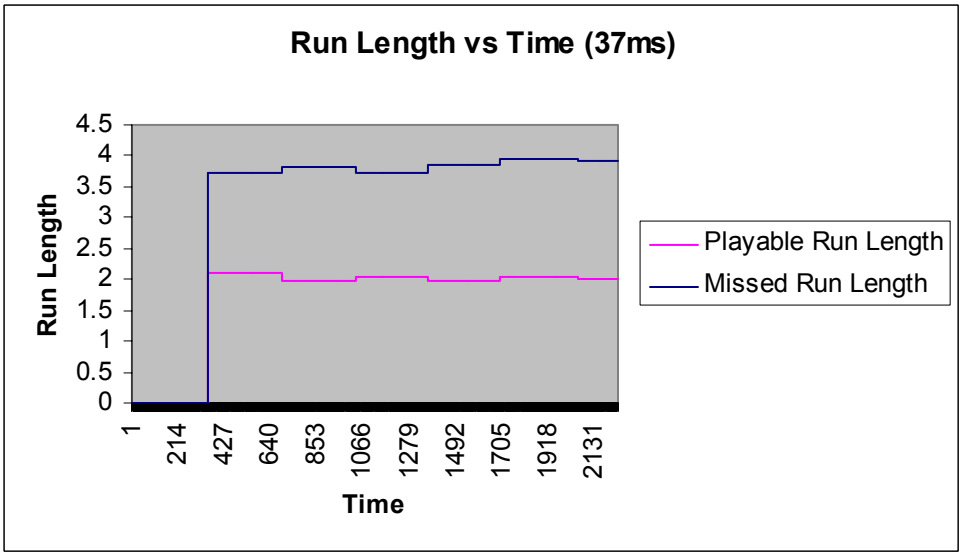

Figure 8.

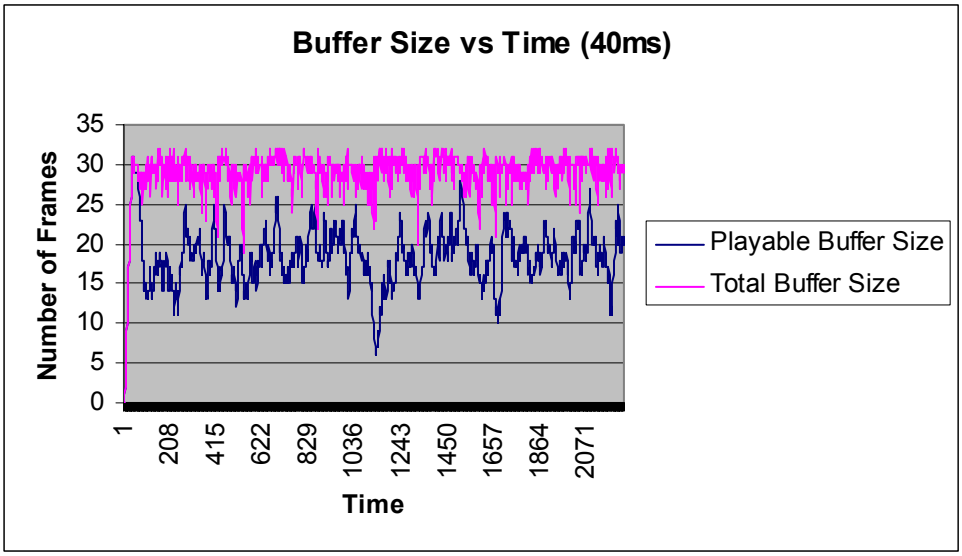

Figure 9.

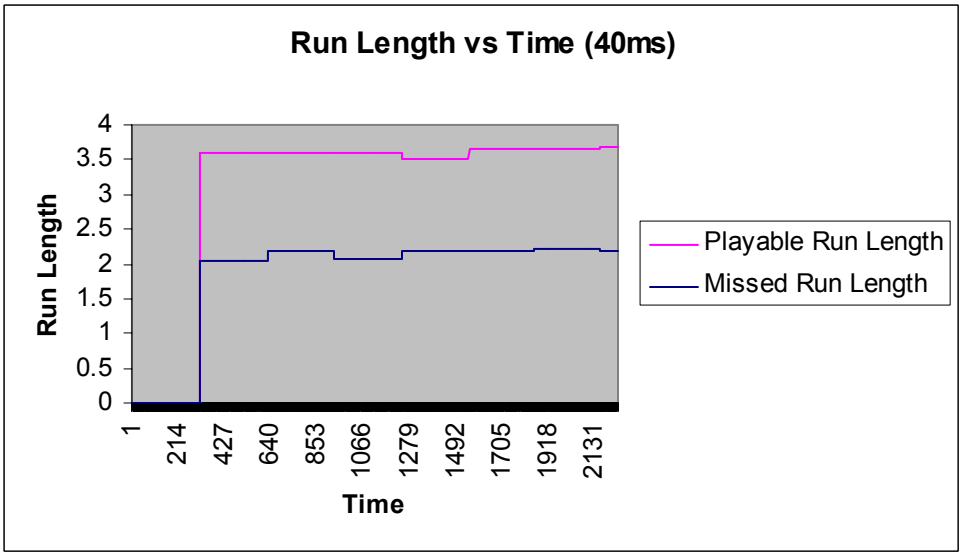

Figure 10.

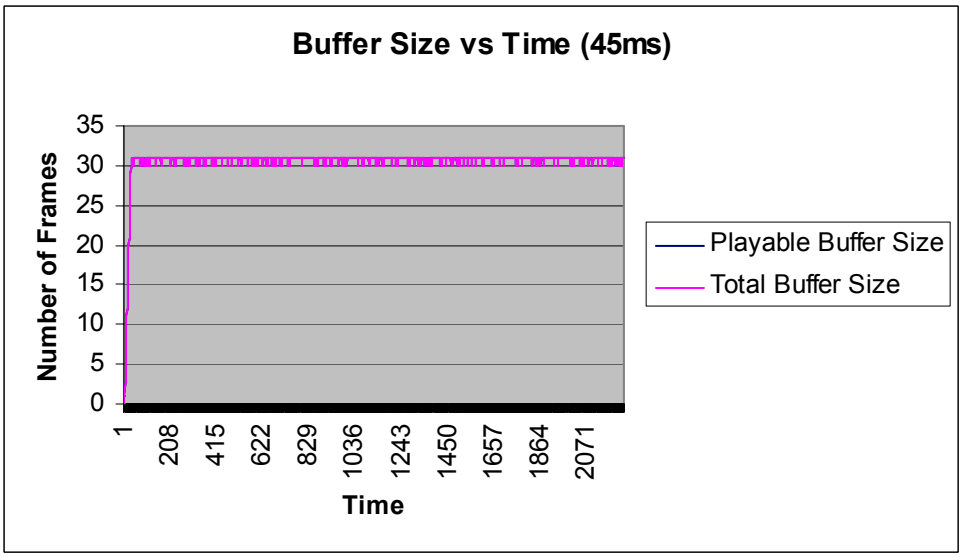

Figure 11.

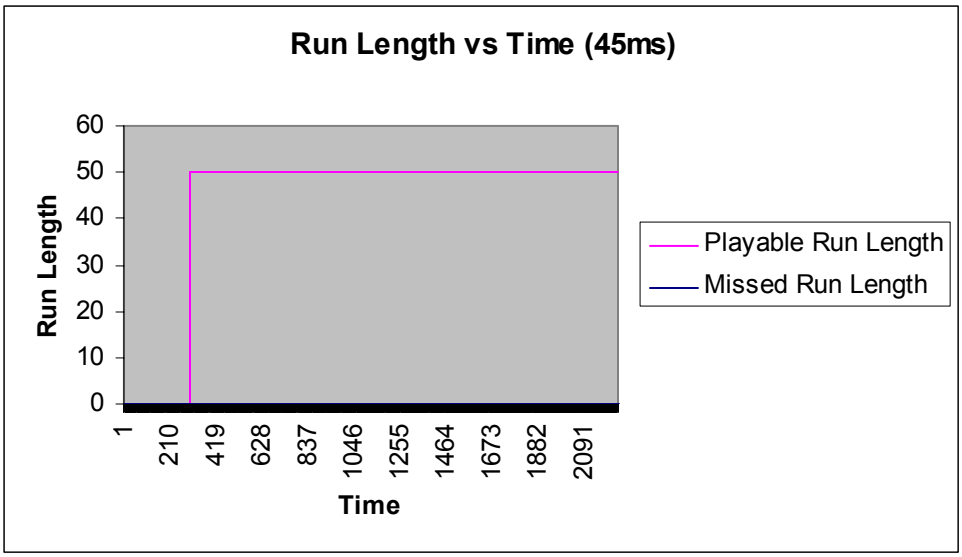

Figure 12.

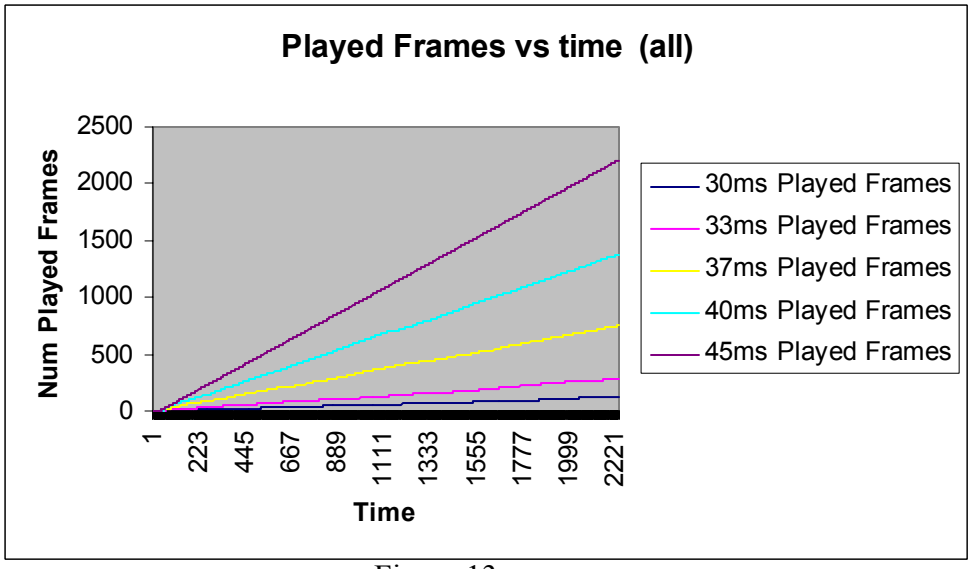

Figure 13.

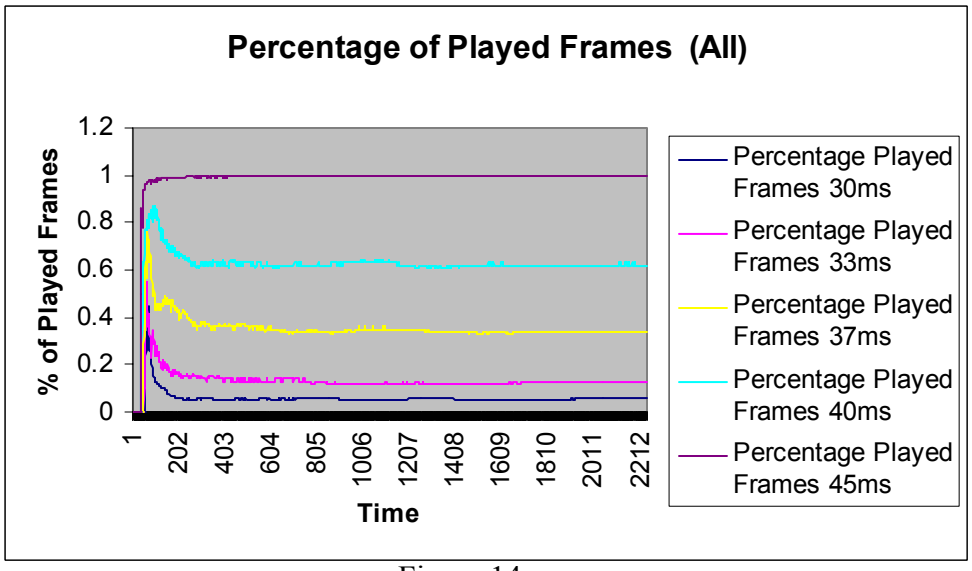

Figure 14.

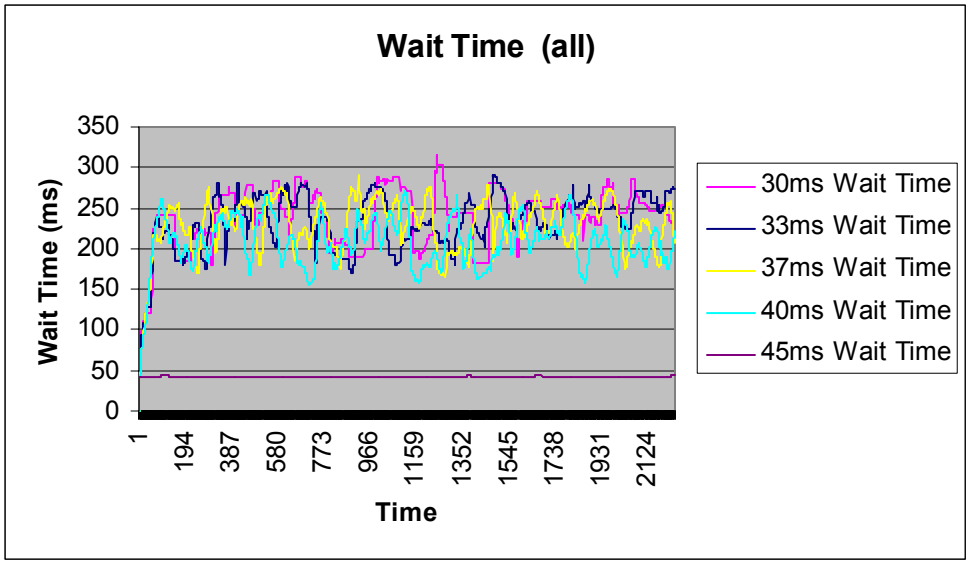

Figure 15.

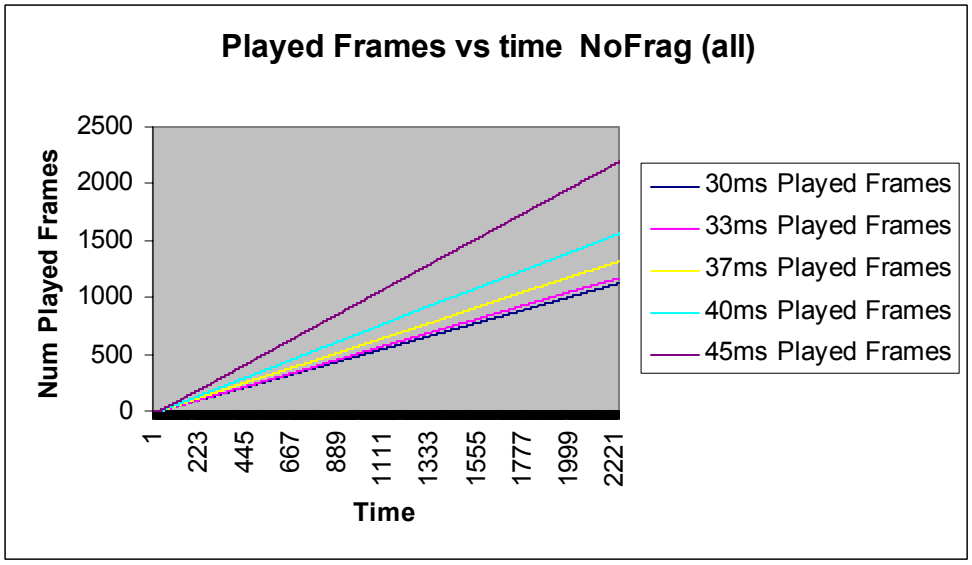

Figure 16.

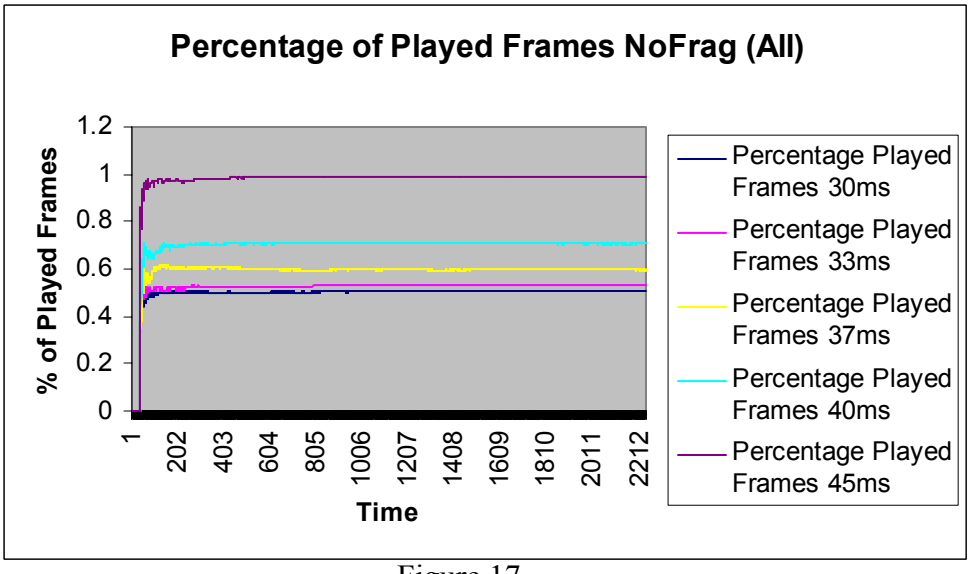

Figure 17.

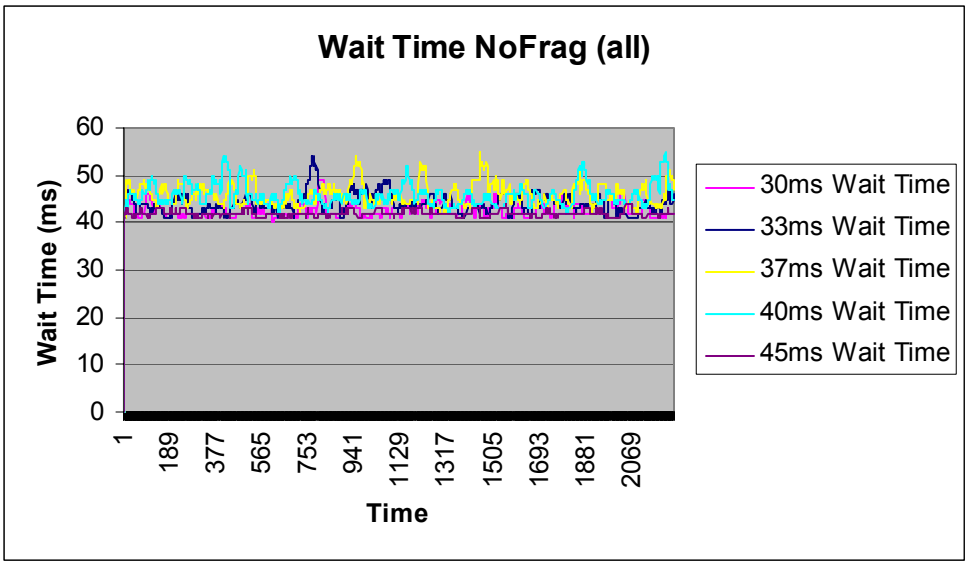

Figure 18.

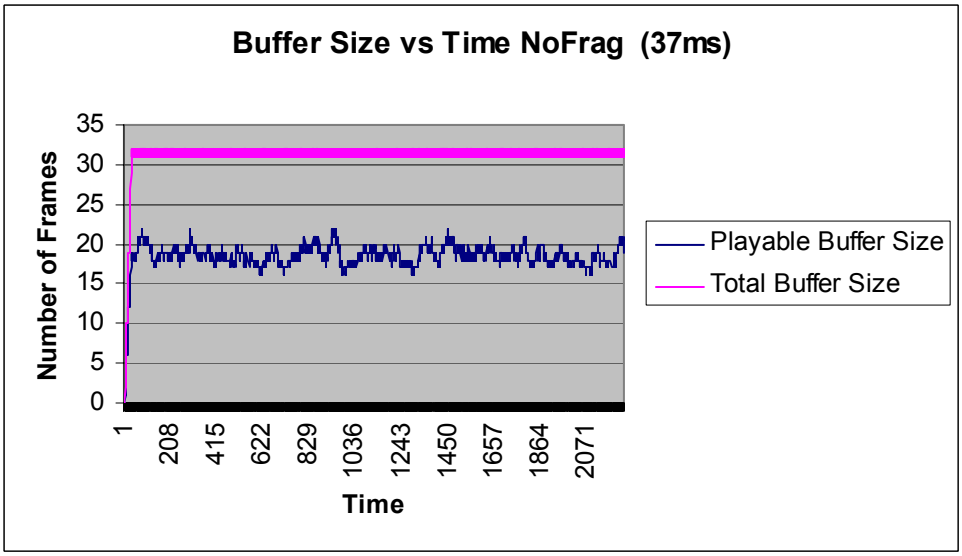

Figure 19.

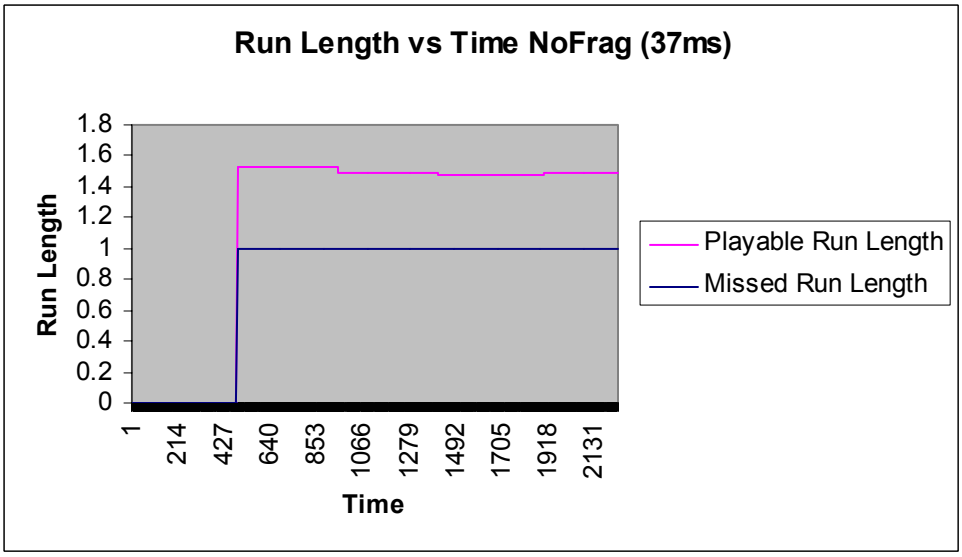

Figure 20.