LIVE VIDEO STREAMING FOR HANDHELD DEVICES OVER AN AD HOC NETWORK

by

PIYUSH MANDOWARA

B.E., R.G.P.V. BHOPAL 2005

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. Gurdip Singh

Abstract

A streaming video application allows a sequence of “moving images” to be sent over the Internet and displayed by a viewer as they arrive. This application is meant for viewing live videos on handheld devices such as PDAs and iPAQs. It captures video data from a webcam installed on a tablet pc, which is then sent over a UDP socket to an iPAQ via an ad hoc network where live video can be viewed in real time. This is achieved by sending video data, frame by frame, and displaying on iPAQ as it arrives. This application also allows taking a snapshot of the video which can be saved for later viewing and also allows the user to dynamically change the resolution of the video as being viewed.

Two versions of this application have been developed, one using a TCP connection for video transfer between a tablet pc (server) and an iPAQ (client) and the other using a UDP connection. This report studies the trade-off between distance and time as each frame arrives at the client for both the versions. This implementation also supports multiple clients to connect to the server and allows video to be viewed simultaneously on more than one client and thus studies the trade-off between distance and time for multiple clients. This project is implemented using C#.NET on Microsoft Visual Studio 2005. It uses Microsoft .NET framework 2.0 for server (tablet pc) and Microsoft .NET Compact Framework 2.0 for client (iPAQ).

Video streaming is useful in several areas such as entertainment media, live conferences, surveillance and security field. For entertainment media, streaming video avoids having a web user wait for a large file to be downloaded before viewing the video. Instead, the media is sent in a continuous stream and is played as it arrives. For surveillance purposes, the streamed video gives a real time view of the field. The primary application of this implementation is to be used in the field of sensor networks.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my major professor and advisor Dr. Gurdip Singh for his constant guidance throughout this project.

I would also like to thank Dr. Daniel Andresen and Dr. William Hsu for being the part of my committee.

CHAPTER 1 - Introduction

Communication and entertainment are the fields where video is considered to be very important media for many years. In olden days when video was used in analog form was totally different from now a days. With the invention of digital circuits and computers video has been totally transformed into digital form and this has led to revolution in communication of video. Moreover popularity of Internet led to communication of video over the network. This kind of communication has lot of complications like low bandwidth, packet loss, and numerous delays. This paper reports such challenges that make simultaneous delivery and streaming of video difficult and implements a system that enables streaming of live video over an ad hoc network, and with the increasing capabilities of handheld devices which makes them popular for delivering multimedia content, provides an opportunity to build such a system for handheld devices.

The high-speed wireless has helped a lot in mobility and increased the flexibility in accessing the Internet. This has also led to mobile devices being part of that Internet. The computation power of mobile devices is so less than a normal computer machine that application meant for computers can’t be directly used on mobile devices. This is because CPU speed, low computation power of handheld devices, size of memory, small display screen etc. Because of these limitations these devices can’t be used for advanced applications. However, now days their capabilities have increased to an extent which makes them capable enough to run big and advanced applications, video streaming application is one of them. Mobile devices like handheld devices, pocket pc etc are now not only used for voice communication but also provides user with various options of running application accessing Internet, checking emails, watching videos over Internet. Thus implementing a streaming video application for handheld devices is a great idea to watch video over mobile devices with the facility of being mobile.

Keeping all these things in mind we have developed a solution to stream video over ad hoc network for handheld devices like iPAQ which is a pocket pc and capable of running Windows Mobile 5.0 operating system. These devices can support special .NET framework known as compact framework which gives capability to run an application built in .NET. Thus they show sufficient potential to act as client for streaming video. This paper talk about all know-how's about streaming video on handheld devices. Next Chapter talks about challenges faced to stream a video. The protocol used to achieve video streaming is described in Chapter 3. Chapter 4 talks about implementation aspects like architecture of system, capturing live video transmitting it over network. The analysis of this implementation is shown in Chapter 5. Finally, Chapter 6 constitutes conclusion and future work in this field.

CHAPTER 2 - Challenges in Video Streaming

This section introduces some of the basic approaches and key challenges in video streaming. The three fundamental problems in video streaming are briefly highlighted and are examined in depth in the following three sections.

2.1 Introduction to Video Transfer Mechanisms

With the advancement in computing technology and the extensive use of multimedia all around the globe, sending and receiving the Real-time multimedia files over the Internet has now become a predominant requirement. New mechanisms have been introduced to allow the video transfer and out of those the most common approaches are the file download mode and the streaming mode. The following sections introduce these approaches and compare them on the basis of the pros and cons of each of them.

2.1.1 Video File Download Mode

The most straightforward approach for video delivery over the Internet is using the download mode or the file download mode. This modes works the same way a normal generic file download works but the only difference in this case is that the file is a large video file rather than a generic file. In this mode, a user downloads the entire video file and then plays back this file. The nature of this mechanism is an HTTP progressive download. The disadvantage of using this approach is that the video content is loaded sequentially i.e. the user cannot access the alter part of the video until the entire video download is complete. Also, the download mode usually takes long and unacceptable transfer time. Since, the video files are usually large in size so they also require large storage spaces. Thus, limitation of storage spaces and transfer times makes this mechanism unreasonable to use.

2.1.2 Video Streaming Mode

Video Streaming is a mechanism where data flow in a digital stream from a server computer to the user's computer. Since the data comes in the form of digital bits the mode is termed as Streaming. This mode gets rid of the bottlenecks of the former one by allowing the user to view the contents of the video as soon as some part of the video gets streamed instead of making the user to wait till the entire video download. There is just an initial wait for a few seconds for a buffer to be built up that helps avoid interruptions caused by Internet traffic problems. Thus, Video delivery by video streaming attempts to overcome the problems associated with file download, and also provides a significant amount of additional capabilities.

The capability of letting the receiver start viewing the video without having the entire video delivered is achieved by chopping the video into parts, each part is then transmitted from the server to the receiver in succession, once the receiver receives the first part it decodes it and can start playing back the video. The receiver continues this process of decoding and playing until all the parts are delivered [1].

2.1.3 Comparing File Download and Video Streaming Mode

The sections above discuss different mechanisms for the video transfer with the advantages of each of them. But it is very important to compare these mechanisms on the basis of some following common factors:

Scalability: The video streaming involves a streaming server which requires a specific, or fixed, allocation of bandwidth between the receiver and the streaming server. This server is capable of supporting only a defined number of users, determined by the total available bandwidth divided by the bandwidth allocated to each user's video stream. Thus, this mechanism puts a limit to the number of users streaming the video at a time. On the other hand, video download mode is similar to a file being delivered from a web server, there is no specific bandwidth allocated to a viewer. The more the number of users is performing the download, the slower the download will be resulting in delay in start time i.e. buffering or loading. Because the progressive download is really just the video file delivery, it is perfect for existing content delivery networks and, therefore, for supporting a very large viewing audience.

Ownership of content: Video streaming provide a form of digital rights management i.e. DRM capabilities providing some degree of confidence that the published content cannot be stolen and passed on to the Internet. The reason for this is that it gets difficult to download or cache raw content from the streaming server. In contrast to this, the video download mode offers limited protection and puts no difficulty for user steal the content from the server.

Cost: Video streaming requires the use of high-cost servers while the low-cost nodes are used for video download mode. Video streaming has expenses associated with bursting. With the file download mode we can plan closer to your peaks.

2.2 Video Streaming as a Sequence of Constraints

The important goal of video streaming is to perform the streaming in a manner so that this sequence of constraints is met. The problem of sequence of constraints is shown below:

Considering a specific example, lets denote the time interval between frames be denoted by X, e.g. X is 33 ms for 30 frames/s video and 100 ms for 10 frames/s video. When each frame is received by the user it must be delivered in sequence of deadlines. Frame N must be delivered by time TN, Frame N+1 must be delivered by time TN + X, Frame N+2 must be delivered by time TN + 2X.

The particular frame that is lost during this transmission is not received at the receiver side and is considered out of sequence or lost. Furthermore, any data that arrives late is also useless. Thus, the video streaming must perform the streaming in a manner so that this sequence of constraints is met. Figure 2.1 illustrates video streaming as a sequence of constraints.

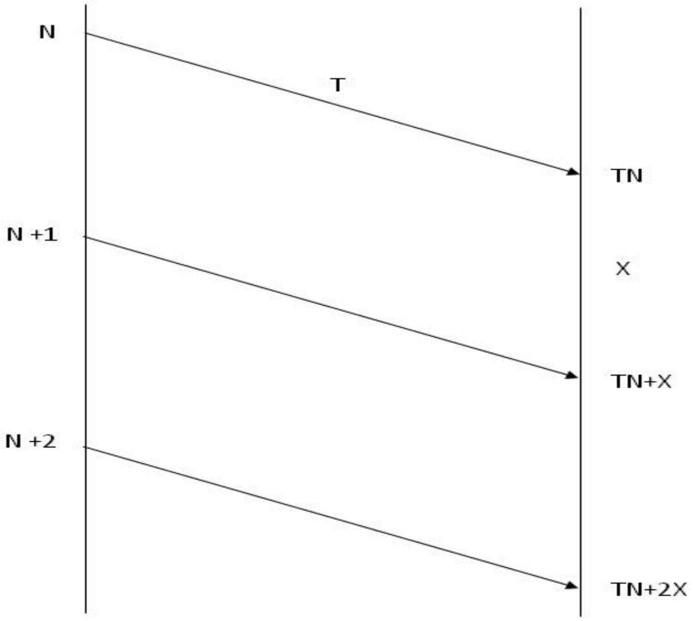

Figure 2.1 Video Streaming as a sequence of constraints.

2.3 Other Problems in Video Streaming

Other problems that affect Video Streaming are:

- Bandwidth
- Delay jitter
- Loss rate

Available bandwidth between two points in the Internet can be time-varying is generally not known. Bandwidth problem occurs when sender sends at a faster rate faster than available bandwidth resulting in congestion, lost packets and degraded quality of video.

When Video Streaming using packet over network then it is possible to receive them out of order so sometime delay in packets results to jitter as they are shown later than their following packet.

High resolution video streaming may result in high loss rate and in the ad hoc network as distance is increased signal strength is low and bits get corrupted resulting in loss of packets.

CHAPTER 3 - Transmission Protocol

This chapter discusses different Transmission Control Protocols used by the Transport Layer of the TCP/IP suite. It includes the basic conceptual understanding of TCP/IP protocols and processes to aid in the configuration, deployment, and troubleshooting of the Microsoft implementation of TCP/IP in Windows NT version 4.0.

3.1 TCP/IP and UDP

TCP/IP and UDP are transport layer protocols used for communication over network. TCP is to pass control messages while UDP is to send video data.

3.1.1 The TCP/IP Protocol Suite

Industry-standard TCP/IP is an suite of protocols designed for big internet TCP/IP was developed by DARPA. It provides user with high-speed communication network links.

3.1.2 TCP/IP Protocol Architecture

TCP/IP Protocol Architecture is viewed as a layered model which is composed of a four-layers and this model is known as DARPA model. The four layers of this DARPA model are: Application Layer, Transport Layer, Internet Layer, and Network Interface Layer. Each layer in this model corresponds to one or more layers of the seven-layer Open Systems Interconnection (OSI) model [2]. Figure 3.1 illustrates the TCP/IP Protocol Architecture suite. Like in the OSI model, when the data is sent from the sender to receiver it is first passed down the, and sent up the stack when it is being received from the network from the other end.

Each layer in the protocol stack adds up some control information to ensure proper delivery. This control information is called a *header* and is placed in the front of the data to be transmitted. When the data is sent from the topmost layer to the next layer in the stack, the lower layer considers it as data and adds up a header to its front containing the control information. This layer then sends this complete packet to the next layer which then assumes it to be full data and adds up another header and passes on. The addition of delivery information at every layer is called *encapsulation*. When the data is received at the other end, the reverse happens. Once the receiver receives the data from the network it strips off its header before passing the data on to the layer above. As information flows back up the stack, information received from a lower layer is interpreted as both a header and data.

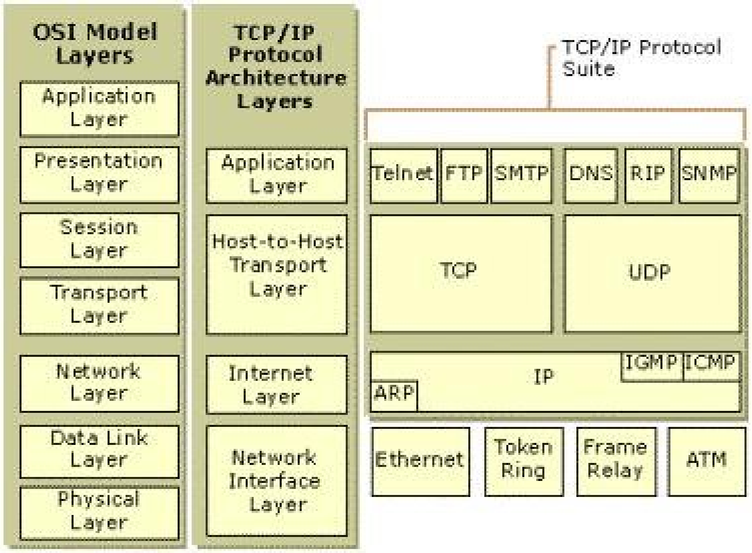

Figure 1.1 TCP/IP Protocol Architecture

3.1.2.1 Transport Layer

This layer is responsible for the delivery of packets between the Internet layer and an Application Layer. The core protocols of this layer are Transmission Control Protocol and User Datagram Protocol. An important concept in the world of TCP/IP communications is that of port numbers. Each application running in the computer will be given a unique port number at the Transport layer. There are two types of protocol within the Transport layer namely, Connection-Oriented Protocol and Transmission Control Protocol. Transport Layer provides two ways of establishing connection between the sender and the receiver namely, TCP and UPD. The features of TCP and UDP are:

- TCP provides a connection-oriented, reliable communications service which is responsible for the establishment of a TCP connection, the sequencing and acknowledgment of packets sent, and the recovery of packets lost during transmission.
- UDP provides a connectionless, unreliable communications service which is used when the amount of data to be transferred is very small and when the overhead of establishing a TCP connection is not desired.

Transmission Control Protocol

TCP is a stream-oriented delivery service where the data is chopped into segments and transmitted from one end to another. TCP segment is treated as a sequence of bytes with no record or field boundaries. The transport of data from one end to another is initiated only after a connection is established between the two ends because of transport layer is called connection-oriented. Each segment in the data transfer is identified by a sequence number which keeps track of the segment being transferred. Encapsulating a sequence number in the segment makes this connection Reliable. When the sender transfers a segment an acknowledgment is used to verify the reception of data at the other end. For each segment sent, the receiving host must return an acknowledgment (ACK) within a specified period for bytes received. If an ACK is not received, the data is retransmitted. The following table states all the fields of the TCP header.

Table 1.1 Key fields in TCP Header

Field	**Function**
Source Port	TCP port of sending host.
Destination Port	TCP port of destination host.
Sequence Number	The sequence number of the first byte of data in the TCP segment.
Acknowledgment Number	The sequence number of the byte the sender expects to receive next from the other side of the connection.
Window	The current size of a TCP buffer on the host sending this TCP segment to store incoming segments.
TCP Checksum	Verifies the integrity of the TCP header and the TCP data.

Transmission Control Protocol Ports

TCP establishes the connection between the two ends at their specific ports. These ports either originate the connection or they can receive new connections. Any port can originate a connection, but the port numbers that receive connections, have to be assigned specific port numbers. A TCP port provides a specific location for delivery of TCP segments. Table 3.2 lists few well-known TCP ports.

Table 3.2 Well known TCP Ports

TCP Port Number	**Description**
20	FTP (Data Channel).
21	FTP (Control Channel).
23	Telnet.
80	HyperText Transfer Protocol (HTTP), used for the World Wide Web.
139	NetBIOS session service.

TCP Connection Mechanism

TCP establishes a connection between the two ends using a three-way handshake mechanism. As discussed in the above section, the sender first establishes a connection with the receiver and then the data transfer can take place. This scenario is established by a three-way handshake mechanism. The purpose of this mechanism is to synchronize the sequence number and acknowledgment numbers of both sides of the connection, exchange TCP Window sizes, and exchange other TCP options such as the maximum segment size. The three-way handshake mechanism is described as follows:

1. The client initiates the connection by sending a TCP segment to the server with an initial Sequence Number and the Window size indicating the size of a buffer on the client to store incoming segments from the server.
2. The server sends back a TCP segment containing its chosen initial Sequence Number, an acknowledgment of the client's Sequence Number, and a Window size indicating the size of a buffer on the server to store incoming segments from the client.

3. The client sends a TCP segment to the server containing an acknowledgement of the server's Sequence Number.

TCP uses a similar handshake process to end a connection. This guarantees that both hosts have finished transmitting and that all data was received.

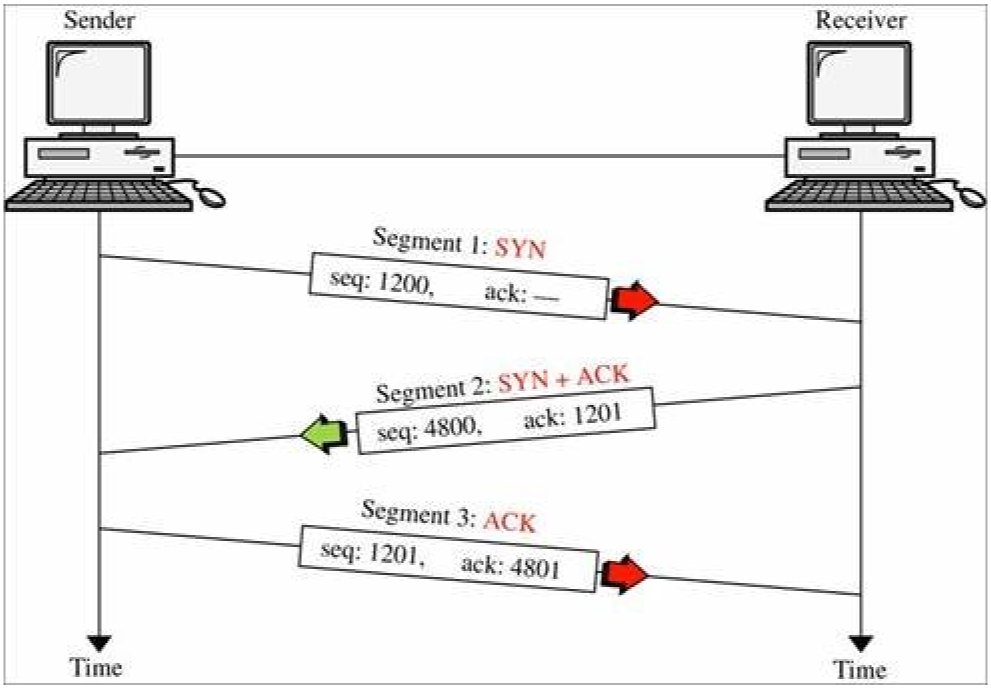

Figure 2.2 TCP 3-Ways Handshake

User Datagram Protocol

Transport Layer introduces another protocol for the transfer of data between the two ends using User Datagram Protocol. It is a connectionless datagram service that offers unreliable, best-effort delivery of data transmitted in messages. UDP lacks any kind of mechanism to control or avoid network congestion or other forms of network-based control mechanisms needed to implement to ensure smooth flow of traffic in the network. This means that the arrival of datagram's is not guaranteed; nor is the correct sequencing of delivered packets. UDP does not recover from lost data through retransmission.

UDP is used by applications that do not require an acknowledgment of receipt of data and that typically transmit small amounts of data at one time.

Table 2.3 Key fields in UDP header

Field	**Function**
Source Port	UDP port of sending host.
Destination Port	UDP port of destination host.
UDP Checksum	Verifies the integrity of the UDP header and the UDP data.
Acknowledgment Number	The sequence number of the byte the sender expects to receive next from the other side of the connection.

User Datagram Protocol Ports

UDP ports allow application-to-application communication between sender and receiver. An application must supply the IP address and UDP port number of the destination application. A port provides a location for sending messages. The port field is a 16 bit value, allowing for port numbers to range between 0 and 65,535. Port 0 is reserved, but is a permissible source port value if the sending process does not expect messages in response. Table 3.4 lists few well-known UDP ports.

Table 3.4 Well known UDP Ports

UDP Port Number	Description
53	Domain Name System (DNS) Name Queries.
69	Trivial File Transfer Protocol (TFTP).
137	NetBIOS name service.
138	NetBIOS datagram service.
161	Simple Network Management Protocol

TCP/IP Application Interfaces

Windows NT makes industry standard *application programming interfaces* (APIs) available so that application can access core TCP/IP services.

Figure 3.3 shows two common TCP/IP application interfaces, Windows Sockets and NetBIOS, and their relation to the core protocols.

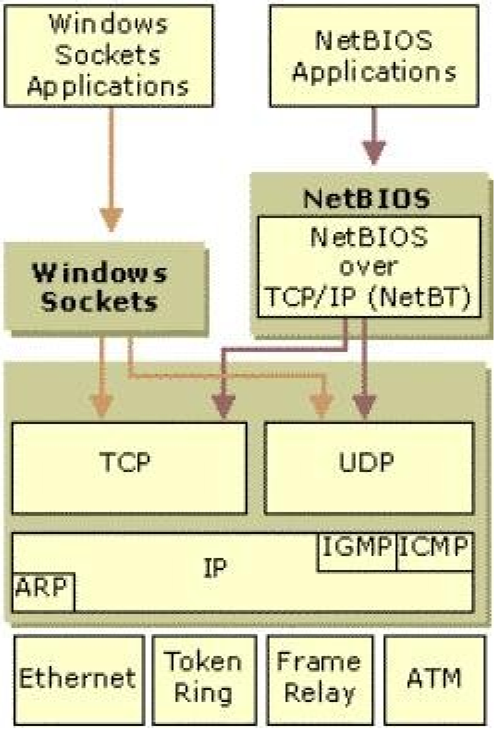

Figure 3.3 TCP/IP Application Interfaces

Microsoft .NET specific classes

3.2.1 TCPClient

TCPClient class is used for sending and receiving data by means of TCP protocol. The data is in form of stream and thus it can be sent and received by .NET Framework stream-handling techniques. As the TCPClient uses TCP protocol, there must be a TCPListener listening to a socket.

3.2.2 TCPListener

TCPListener listens for an incoming request from a TCPClient class at a given TCP port. To start listening *Start* method is used and stop method is used to *Stop* listening. The start method will queue each incoming request till either the stop method is called or the queue reached its limit for maximum connection. There is a queue for incoming connection request from client. *AcceptSocket* or *AcceptTcpClient* is used to pull one of the request from this queue. It is important to note that the stop method does not close any existing connections.

3.2.3 UDPClient

UDPClient class is used for sending and receiving data by means of UDP protocol. As UDP is connectionless there is no need to establish a connection with the remote host before sending or receiving data. This class also has methods which allow transfer of multicast datagrams.

3.2.4 NetworkStream class

NetworkStream is used to transfer data in synchronous or asynchronous mode. The data is send and received using stream sockets. A connected socket must be first established before network stream can be used. In case the connected socket is using file permission, then these should also be specified. By default, even if the network stream is closed the socket remains open. However, to change this behavior permission can be given to network stream to close the socket. For this we need to set the value of owns Socket parameter to true.

Read and write can be performed simultaneously on a socket. There is no need of synchronization for this, as far as both operations are performed through a separate thread. *Read* and *Write* methods are used to perform synchronous blocking I/O with just a single thread. To make use to separate thread for both operations *BeginWrite/ EndWrite* and B*eginRead/EndRead* methods are used.

3.2.4.1 GetStream

GetStream is one of the methods implemented by the TcpClient Class which returns a NetworkStream used to send and receive the data. A rich collection of methods and properties can be applied on this NetworkStream to facilitate network communications.

The transfer of data is initiated first giving a call to Connect method else GetStream method will throw an *InvalidOperationException*. Once we have obtained the NetworkStream, call the Write method to send data to the remote host. Call the Read method to receive data arriving from the remote host. The Read and Write methods get blocked block until the specified operation is performed but this blocking on Read method can be avoided by checking the *DataAvailable* property to true which means that data has arrived from the remote host and is available for reading. This guarantees that the data has been read completely. Everytime a NetworkStream is created it must be closed as closing TcpClient does not release the NetworkStream .

Asynchronous Server Socket & Calls

Asynchronous sockets are non-blocking sockets. When reading from a socket can be in blocking to wait until data is there to read, asynchronous sockets allows not to block program for data to come instead it will undergo its normal execution and whenever data is available to read a call back method is used which reads data from the socket.

CHAPTER 4 - Implementation

This chapter details the key elements of the video streaming project put forth in this report.

4.1 Ad hoc Network

An ad-hoc network is dedicated network established to connect network devices on the fly. It gives appearance of connection between devices as they are connected over Internet. Ad-hoc also known as spontaneous network is a LAN or other small network, in which network devices are part of the network only for the duration of a communications session especially one with wireless or temporary plug-in connections, or, in the case of portable or mobile devices, when they stand in close proximity to the rest of the network. In Latin, *ad hoc* literally means "for this," i.e. “for this purpose only," and therefore are established on the temporary basis for limited time in use and thus not concrete.

There has been lot of work done as far as ad hoc networks are concerned, lot of projects are only meant for this kind of network to keep them isolated from rest of the world, also to provide security against outside threat when data transferred is highly sensitive. Like for the purpose of military and defense and related to government. Internet Protocol suite was inspired by packet radio system which were one of the earliest wireless ad hoc network. Thus projects funded by DARPA for defense utility and others made this field as significant thing in life.

The network which lack fixed infrastructure and pre-configuration characterize the term ad hoc network which can dynamically communicate, propagate and organize. The performance of conventional networks highly depends on bandwidth but ad hoc network doesn’t also regular old networks depends on continuous connectivity, reliable power supply and topology and static configuration. An ad hoc network does not rely on these supporting infrastructures. As suggested by Murphy in his paper on formal reasoning for mobile communications, an ad hoc network can be visualized as a continuously changing graph. Communication depends not only on the distance between nodes, but also on the willingness of individual nodes to collaborate and form a cohesive transitory community. A wireless ad hoc network is formed on the fly, and changes as the nodes enter, relocate or leave the network.

4.2 Device Specifications

Devices used for this project are stated as under:

Server - ASUS R2H Ultra Mobile PC
Windows XP Tablet PC Edition Operating System
Acts as Server

Client - HP iPAQ 2790
Microsoft Windows Mobile 5.0 Operating System
Acts as Client

4.3 System Architecture

The architecture for Video Streaming consists of three layers of operation at Server side and two layers of operation at client side. At server side the first two layers Webcam Driver and Video Capture are responsible for making the video captured by webcam available to Streaming Server which listens to incoming connections from client and streams video on request.

At client side Streaming Client receives the streamed video and Video Display layer takes care of showing the video on the screen.

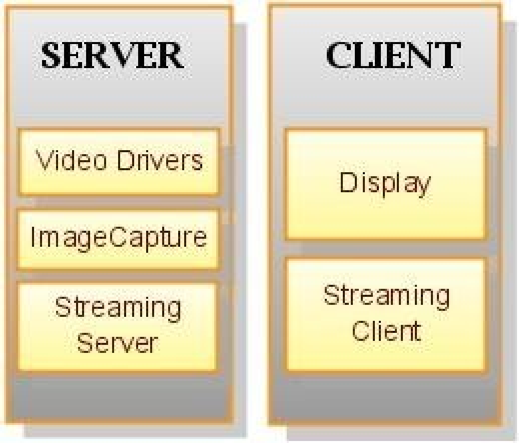

Figure 4.1 Layered Architecture

First step in streaming a video is to capture the raw video produced by webcam. It requires accessing the drivers of operating system which helps in communicating with an external hardware device. Webcam is connected to the streaming server which runs on Microsoft Windows XP so the drivers used are defined for Windows XP Operating System. Live feed of video captured by webcam is first previewed in the picture box control of the form on server ASUS R2H Ultra Mobile PC. The video previewed is saved as series of individual images after converting it to a bit-stream and made available to the server. As the client request for connection comes to the server listening on a designated port number followed by a request for video, the series of saved image object stored as bit-stream are sent over the an ad hoc network to client. This process continues until client sends a request to STOP video streaming.

At client side the received bit-stream is converted to an image frame and displayed in the picture box control on the form suitable to fit in the screen of iPAQ. Displaying the series of continuous incoming images on iPAQ is similar to displaying a video and achieves the primary objective of video streaming. Complete architecture of the system is shown in the figure below.

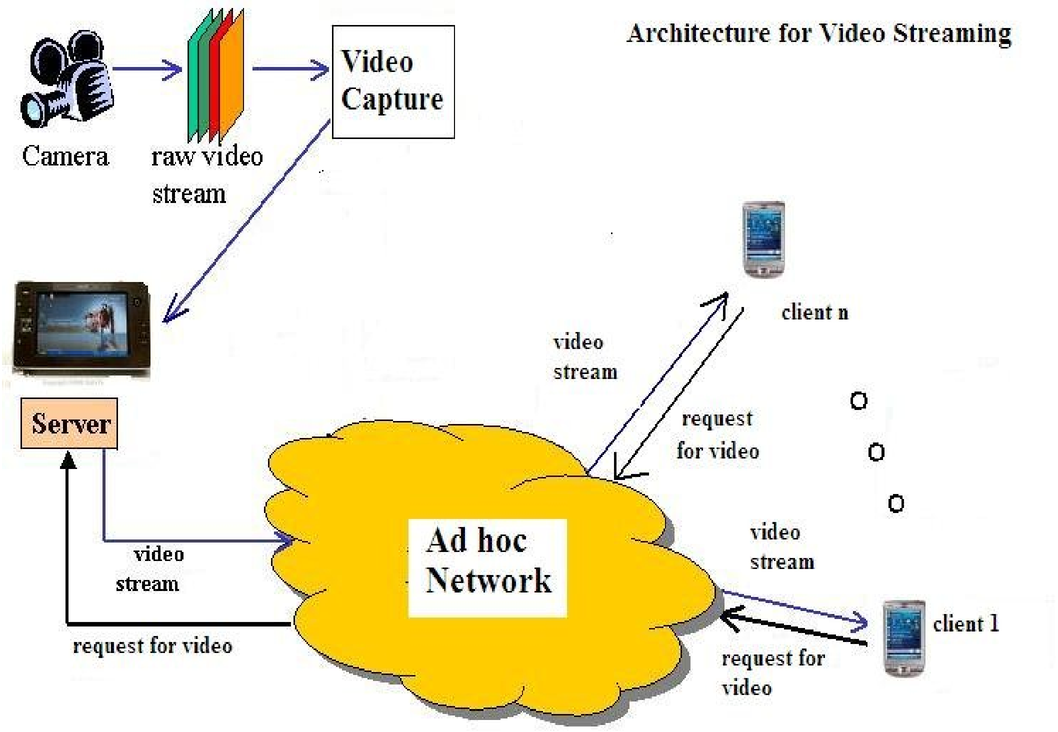

Figure 4.2 System Architecture

4.4 Interfacing with Webcam

Webcams are web cameras usually used to send images over WWW World Wide Web by the use of some instant messaging service. Webcams are usually low-resolution cameras which can be used as digital cameras to take pictures and video when connected to your pc. Here we need to interface the webcam with our image capture program which can capture live images from webcam. That means it should be at server end. Here in this case our server is ASUS R2H which has inbuilt Asus Webcam so we don’t need to plug it in every time when we want to capture images.

For Video Streaming webcams are like camera connected directly to a PC. To integrate a webcam in our application Windows provide support by providing various drivers to access live feed from an external device. Next section will elaborate on this that how we can integrate a webcam in our application by using device drivers and windows operating support.

4.4.1 Program the Webcam

Programming a webcam requires support from OS as it an external hardware device whose output we need to grab in our application. For this Windows provides AVICap class API. This class is Audio Video Interleaved Capture class to capture audio or video from external device. According to scope of the project we will only concentrate on capturing the video from external device which is webcam in this case.

4.4.1.1 AVICap

The AVICap class can be found in the avicap32.dll file provided by Windows which contains waveform-audio acquisition hardware and message-based interfaces to access video and, provides the ability to capture streaming video to disk. The only problem in using AVICap class is that we have to use Platform Invoke (P/Invoke) to use the API as it is a Win32 API and is therefore not exposed as a managed class.

It supports capturing of streaming video and real time capture of single-frame. Also it provides control of devices like Media Control Interface (MCI) devices video sources that a user can control to start and stop positions of a video source, and augment the capture operation to include step frame capture.

For capturing the video in our window using AVICap we have to first capture video streams to an audio-video interleaved (AVI) file then connect and disconnect video input devices dynamically then view a live incoming video signal by using the overlay or preview methods and then specify a file to use when capturing and copy the contents of the capture file to another file.

We then set the capture rate and display dialog boxes that control the video source and format then copy images to the clipboard and then capture and save a single image as a device-independent bitmap (DIB).

Programmatically capturing the video from a webcam requires initialization of following constants before using AVICap dll. These are the commands sent to the captured window to obtain required functionality. The captured windows interprets the commands by the initialization value given to that command of avicap.dll.

```
WM_CAP_START = 1024;
WS_CHILD = 1073741824;
WS_VISIBLE = 268435456;

WM_CAP_DRIVER_CONNECT = (WM_CAP_START + 10);
WM_CAP_DRIVER_DISCONNECT = (WM_CAP_START + 11);
WM_CAP_EDIT_COPY = (WM_CAP_START + 30);
WM_CAP_SEQUENCE = (WM_CAP_START + 62);
WM_CAP_FILE_SAVEAS = (WM_CAP_START + 23);

WM_CAP_SET_SCALE = (WM_CAP_START + 53);
WM_CAP_SET_PREVIEWRATE = (WM_CAP_START + 52);
WM_CAP_SET_PREVIEW = (WM_CAP_START + 50);

SWP_NOMOVE = 2;
SWP_NOSIZE = 1;
SWP_NOZORDER = 4;
HWND_BOTTOM = 1;
```

4.4.1.2 Creating a Capture Window

To capture a window we need to use *capCreateCaptureWindowA* function. It returns a *handle* of the new window created. It returns a 32-bit integer referred to as handle used to reference an object which is the window in this case.

To achieve future functionality the commands are sent to this newly created window which interprets this commands and do the needful. These commads are sent with SendMessage function which is built into the Win32 API. Whenever we call this function to give commands to new window we pass the handle to that window.

4.4.1.3 Connecting the Window

We now connect this newly created window for capturing video. We connect this to video driver which is installed with video camera. To connect window to driver we use `WM_CAP_DRIVER_CONNECT message.` What is does is simply binds the window to the video driver usually indexed as 0 which is the first driver it finds. This command to connect window can be sent to any window but only capture window can understand it.

4.4.1.4 SendMessage Function

It is used to send commands to capture window to achieve desired functionality out of it. It calls procedure of window and will wait until procedure of window has finished processing the command.

The SendMessage API is passed – *hWnd* a handle to a window, *wMsg* message to send to that window, a short parameter *wParam*, and a long parameter *lParam*. The *w* in *wParam* stands for "WORD" and the *l* in *lParam* stand for "LONG".

For example following sendmessage function call connects the capture window with the handle *hWnd* to the MSVIDEO driver and then can be disconnected by calling *capDriverDisconnect* macro:

```
SendMessage(hWnd, WM_CAP_DRIVER_CONNECT, 0, 0)

capDriverDisconnect(hWnd)
```

4.4.2 Saving the Video as Images

In order to stream video over the network we need to provide server with sequence of images captures before that we need to save it every time we capture so that server can use it anytime when client request comes. As we can now display image in our application we just take a snapshot of that image every 100 millisecond and copy it to clipboard. From clipboard we then convert it to image object and save it to memory stream in a bit map format. Then we extract the image as an array of bytes from memory stream. There is global variable as Image to which we save this array of bytes. This variable is made available to server to send it over network. Sending sequence of such images will result in streaming video.

4.5 Client-Server Architecture

A client program first opens a TCP connection to a server listening on a designated port. The client then passes information about itself as well as some instructions to the server using a command protocol. After the client has passed this information to the server, the client issues a command that tells the server it is ready to accept the video data. The server then sends the data in a rate-controlled way to the client using a 2nd protocol based on UDP. Finally, as displays the data as it receives the UDP packets.

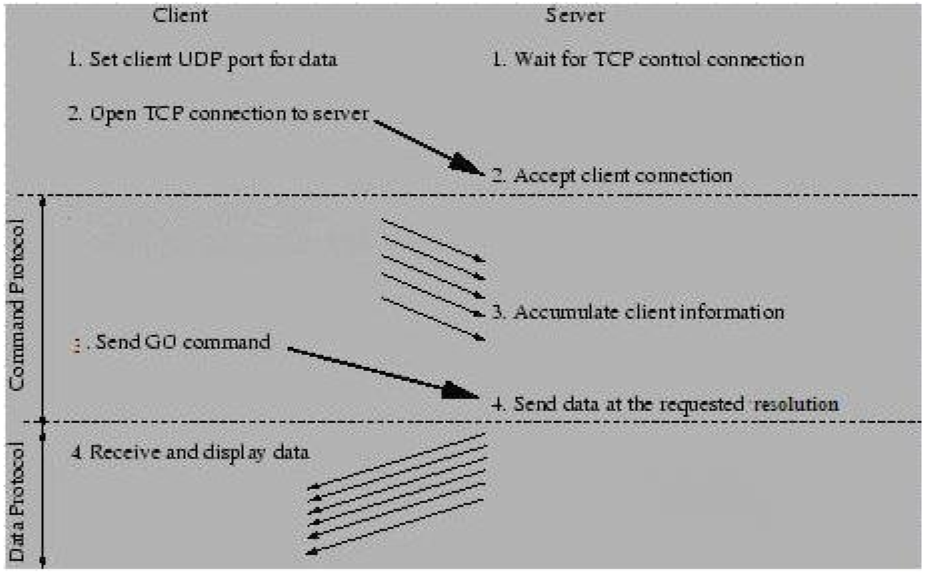
Client
Server
1. Set client UDP port for data
1. Wait for TCP control connection
2. Open TCP connection to server
2. Accept client connection
Command Protocol
3. Accumulate client information
3 . Send GO command
4. Send data at the requested resolution
Data Protocol
4 Receive and display data

Figure 4.3 Overview of Client-Server System

4.5.1 Client Construction

Streaming Client is stand-alone program written in C#.NET using Microsoft Visual Studio 2005 and compiled using Microsoft .NET Compact Framework 2.0 for handheld devices.

This client code is designed to run on HP iPAQ 2790 pocket pc. It supports Microsoft Windows Mobile 5.0 Operating System.

4.5.2 Client Structure

Figure shows the basic controls of the client display. There are four buttons, one text box, one label and one picture box. Text box takes IP Address of server as input and Start button will start the preview of video in picture box control by displaying series of continuous images received from server after establishing a TCP connection to server. Up and Down button will increase and decrease the resolution of video respectively. Clicking on picture button will take a snapshot of the video and save the image for later viewing. Start button now turns to resume button clicking on the Resume button will restart the video.

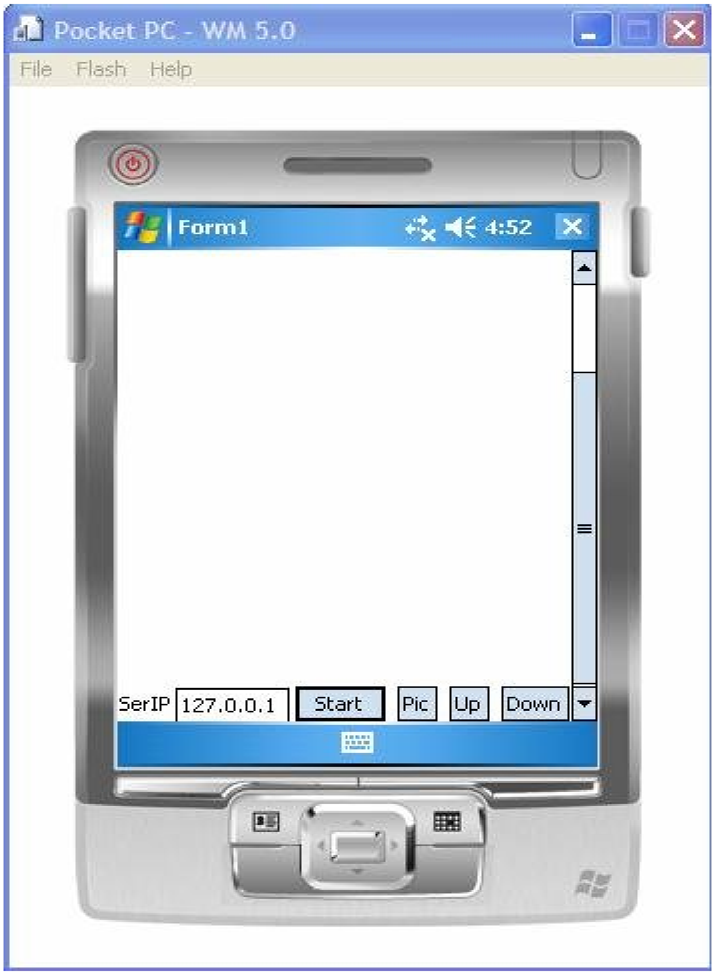

Figure 4.4 Client Display

4.5.3 Client Operation

The client performs the following steps to display a video file. These follow the steps as outlined in Figure.

1. Decides a UDP port to accept data on. This can be any unused port over 1024 on the system.
2. The client opens a TCP connection to the server as user clicks “Start” button using the server IP is as specified by user in text box.
3. Begins the control protocol. In this phase, the client will send the server its IP address, port it is accepting data on and the requested resolution. These are sent using Network Stream.
4. Sends a command to the server to begin sending video.
5. Wait for the incoming data on the UDP port and display it as it arrives.
6. The client sends a stop message to server if user clicks Stop button to stop the video display in picture box and breaks the TCP connection.
7. Takes a snapshot of the video if user clicks “Picture” button and save it for later viewing, at this time video also halts and displays the last image. Video can be resumed by clicking on Resume button.
8. Allows user to change the resolution of video dynamically as the video is being played in picture box. It sends a message to server to increase or decrease resolution from next frame onwards.

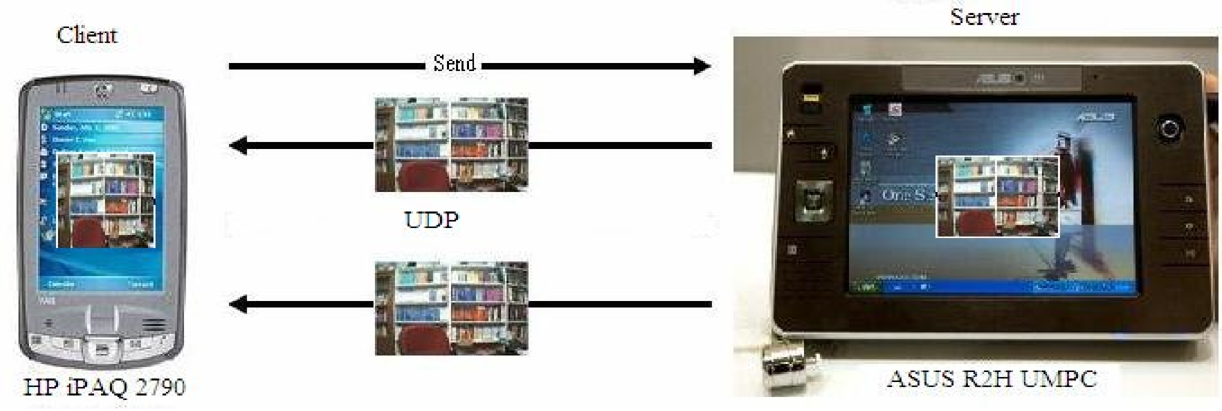

Figure 4.5 Client Operation

4.5.4 Server Construction

Streaming Server is program written in C#.NET using Microsoft Visual Studio 2005 and compiled using Microsoft .NET Framework 2.0. This client code is designed to run on any Windows machine. But in this case we are using Asus R2H Tablet PC to run the server.

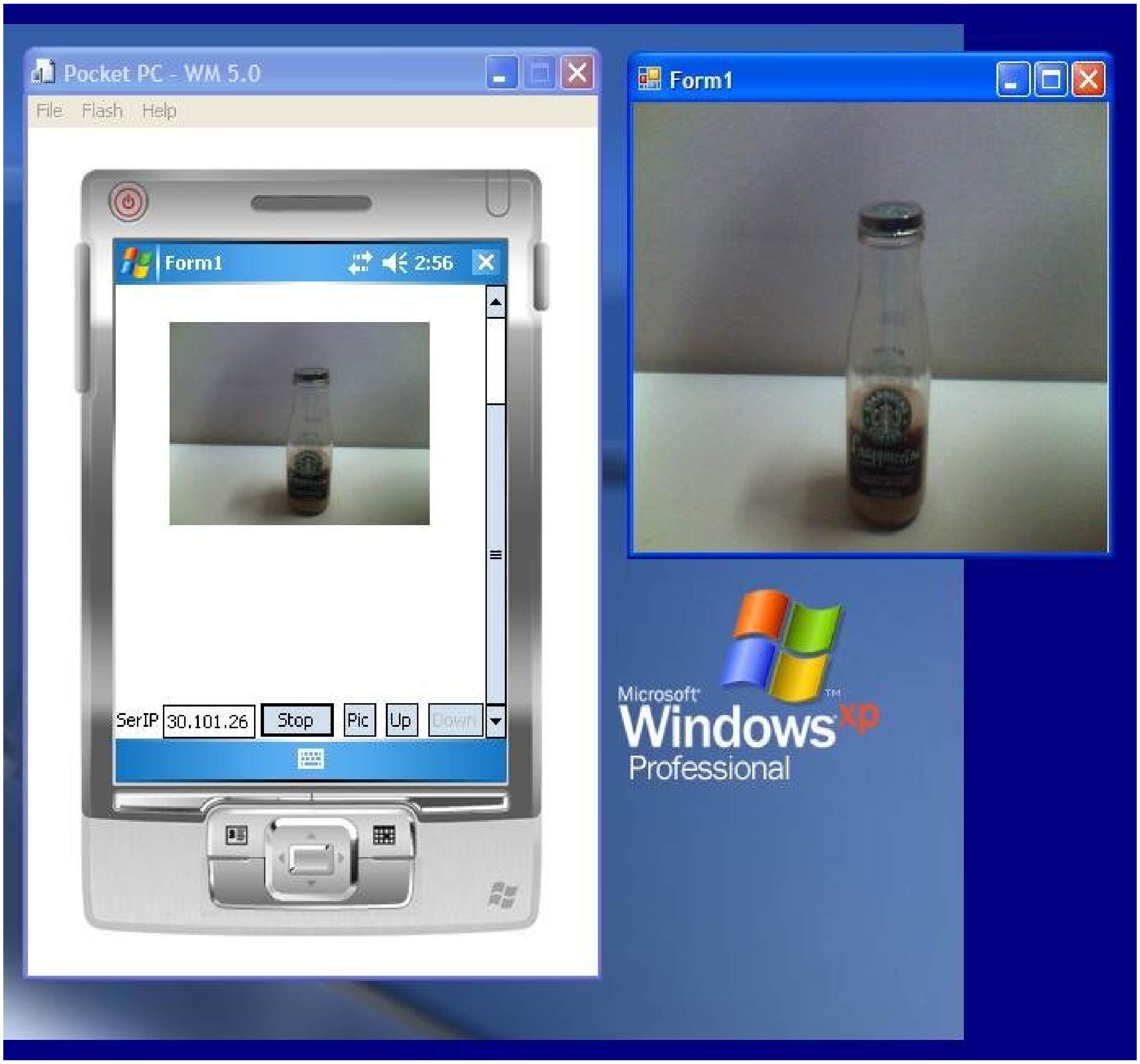

Figure 4.6 Application run on local machine

The main server class is named as VideoServer.cs. Server will accept requests and commands via TCP and output a bit-stream of UDP packets. The basic structure of the server is an infinite loop. Before the loop, the server sets up a socket to listen on the connection. Next, on each loop iteration the server performs the following operations as shown in Figure 4.3.

4.5.5 Server Operation

1. Accepts a client connection on a TCP port. This is the control connection. The server obtains the remote client's IP address via this connection.
2. Processes the client information. The server needs to know the UDP port the client is listening in on and the resolution client wants.
3. Once the Send command is issued, the server begins streaming the video at the requested resolution to the client.
4. The server terminates sending data to the client when it receives Stop command from client or preview of video at server is closed.

In step 1 mentioned above server program creates a *TcpListener* that listens for TCP connection on the specified port. It uses TCPListener method described in Chapter 3. The listening program runs on a separate thread so it doesn’t block the operation of capturing the video from webcam. As connection requests come and gets queued, *TcpListener.AcceptTcpClient* Method synchronously extracts the first pending connection request from the connection request queue of the listening socket, then creates and returns a new Socket and listener can continue to listen on the same listening port. This returned Socket cannot be used to accept any additional connections from the connection queue. However, we can call the *RemoteEndPoint* method of the returned Socket to identify the remote host's network address and port number as mentioned in step 2 while the UDP port number is predefined for all the clients.

In step 3 as server receives Send command issued by client, reading of control messages is done asynchronously. Synchronous read means that the method is blocked until the read operation is complete, and then the method returns its data. With asynchronous read, a user can call *FileStream.BeginRead*. The main thread can continue doing other work, and later the user will be able to process the data. The return type of this method is *IAsyncResult* which is passed as argument to callback method *ReceiveMessage*. *BeginRead* method is called only when data is available to read. After receiving command from client server can process it and start sending the global *Image* variable which stores the image captured from webcam in the form of bit-stream depending on the requested resolution. On receipt of Stop command form client the server stop sending any more images and closes the TCP connection.

4.6 Threading Architecture

Threading helps a program to perform concurrent processing so to perform more than one operation at a time. The *System.Threading* namespace in C# provides classes and interfaces that helps multithreaded programming and allows performing tasks such as creating and starting new threads, synchronizing multiple threads, suspending threads, and aborting threads.

In Client-Server Architecture when multiple clients can connect to server than it is very important to keep shared variables thread safe. So synchronization locks are being used to keep threads accessing shared data which may result in wrong program execution and to fatal results. To provide server capability of handling multiple clients it is important to introduce threads to handle if more than one client connects to server at same time. Instead of making client to wait until other finishes, each client should get feeling that it is being server by server promptly and for that reason multi-threading is used in this project. With multi-threading there are lot of issues related to shared data which can be accidentally modified by another thread so it is important to provide synchronization locks to those shared variable so that only one thread can use it one at a time. This way accidentally modification of common data can be put to an end.

C#.NET provides with multi-threading architecture and to hide shared variable it provides synchronization locks so that only one thread can enter into critical section at a time. Server is C#.NET Windows application in which as the form loads before that all the necessary initializations for capturing a video from webcam is done. After the form is loaded it will call to *VideoDisplay()* function and capture the video in the picture box control of the form. There is a timer on form which takes snapshot of video every 100 milliseconds and stores it as an image in a global variable. This all takes place in main thread, in the form load event there is new thread created to listen for TCP connection on a port.

Following are the code snippets representing threads created for various operations:

```
// thread to listen for incoming connections---
System.Threading.Thread t;

//preview the webcam in picture box control
VideoDisplay(pictureBox1);

//listen for incoming connections from clients
t = new System.Threading.Thread(Listen);
t.Start();
```

After creating a new thread to listen for connection request following code is executed:

```
//listen for incoming connections
private void Listen()
{
System.Net.IPAddress localAdd =
                           System.Net.IPAddress.Parse(IP_Address)
                      ;
      System.Net.Sockets.TcpListener listener = new
                System.Net.Sockets.TcpListener(localAdd, portNo);
      listener.Start();
      while (true)
      {
           VideoServer user = new
           VideoServer(listener.AcceptTcpClient());
      }
 }
```

Function *listener.AcceptTcpClient ()* is a blocking function which will return an object to TcpClient whenever any TCP connection request is received. As server can handle multiple clients so in case of multiple requests at a time accept function will extract the first request from the queue of pending connection requests.

It then creates and returns the handle to new object of TcpClient which is in turn a new socket. After the successful completion of accept returns a new socket handle, the accepted socket cannot be used to accept more connections. The original socket remains open and listens for new connection requests while new returned socket can be used for further communication with that client. After a connection request arrives it creates an object to VideoServer class to which it passes that newly created object of TcpClient, as the constructor is called it gets this new object from which it extracts and initializes all the required information about the client. In the constructor of this class it adds client to the hash table along with the object with which it was called, this object is of VideoServer class.

```
public static Hashtable AllClients = new Hashtable();
AllClients.Add(_clientIP, this);
```

Then it calls the *BeginRead()* function for this client to read control messages if any sent by the client. This is non-blocking asynchronous read which doesn't halts the execution of the program, being asynchronous it will not block the program until there is any data from client to read instead it will continue to perform the normal operation of capturing video and saving it to images and serving to other clients but as soon as there is some data to read, this function will be called which has a call back function as *ReceiveMessage()*.

```
_client.GetStream().BeginRead(data, 0,
(int)_client.ReceiveBufferSize, ReceiveMessage, null);
```

The data can be read inside the call back function by calling *EndRead()* function.

```
 public void ReceiveMessage(IAsyncResult ar)
{
     //read total number of bytes from client
    int bytesRead;

         .
         .
         .
         lock (_client.GetStream())
         {
             bytesRead = _client.GetStream().EndRead(ar);
         }
         .
         .
         .
 }
```

As explained in section 3.3 about asynchronous sockets and calls the *BeginRead()* call back method implements the *AsyncCallback* delegate and returns a void and takes a single parameter of type *IAsyncResult. ReceiveMessage()* is the call back method which implements *AsyncCallback* delegate. This callback method executes in a separate thread and is called by the system after *BeginRead* returns. After getting the Network Stream we make a call to *EndRead* method which returns the number of bytes read as the read operation completes. This read can be shared amongst various threads so it is synchronized by putting a synchronization lock to it.

At Client side there is new thread created for receiving images as follows:

```
     //for receiving images from the server
     private System.Threading.Thread t;

//to begin reading data (images)
     t = new System.Threading.Thread(ReceiveImageLoop);
     t.Start();
```

This will create a new thread for receiving images by calling, this function loops the *ReceiveImage()* function infinitely until main thread calls it off whenever user wants to stop by sending stop message to server.

```
//send the control message
System.Net.Sockets.NetworkStream ns;
lock (client.GetStream())
{
     ns = client.GetStream();
      byte[] bytesToSend =
      System.Text.Encoding.ASCII.GetBytes(message);
      //sends the message
      ns.Write(bytesToSend, 0, bytesToSend.Length);
}
```

4.7 Lines of Code

Layer	Lines of Code
Streaming Server	200
Image Capture	250
Streaming Client	415
TOTAL	**865**

CHAPTER 5 - Testing and Analysis

As we know ad hoc network lacks fixed infrastructure and pre-configuration, and have the capability to dynamically organize and communicate. The performance of conventional networks depends on continuous connectivity, bandwidth, reliable power supply and static configuration and topology. Also the performance of nodes greatly depends on the distances the nodes are away from each other. This will be interesting to study that how the system behaves when client is separated by a distance from the server. Also how the environment to which the ad hoc network is exposed to, can affect the performance of server and client in terms of time difference between each frame received by client and overall performance of system. Further in this chapter we will look at trade-off between distance of server and client, and the rate at which client receives the images also how the indoor and outdoor environment affects the speed of streaming video.

In ad hoc network performance of the system greatly depends on distance between nodes. So we will consider placing server and client 5 meter, 10 meter and 15 meter apart and see how client-server system behaves in terms of speed and robustness. First we will analyze behavior of system when only single client sends request for streaming video to server. Then we will see the case of multiple clients.

There are three different resolution levels a user can choose from while viewing the video. Ofcourse higher resolution will result in slower streaming and lower resolution will result in faster streaming. These three resolution levels are based on standards followed for quality streaming. High quality video streaming means more buffer space and delay in the arrival time of images to clients which degrades the performance and slows the video preview. While low quality will be faster but may not be that soothing to eye so a proper balance needs to be achieved in streaming a video which gives better performance in terms of speed.

Also we will check loss rate in next section. We will make experiment by sending 200 frames from sender and see how many arrives at client side under various combinations of distance between client and server, number of clients single or multiple and with different resolutions.

5.1 Performance Analysis

Following graphs represent frames sent by Sender vs. frames received by receiver. X-Axis represents the sequence number of frames; pink line represents the Sender side frames which are sent to receiver while blue line represents the receiver side frames which are received from sender. Y-Axis represents only two values as Sender and Receiver. Points on straight line corresponding to each sender and receiver represents successful receipt of frame while a sudden drop in this line will tell us about the frames lost and where that drop points to on X-Axis gives the frame number which was not received by receiver.

The tests have been done for a total of 200 frames to observe a trend and get the loss rate of the system. For very high loss rates the sending interval is varied to see if it helps especially for multiple clients at 15 meter range.

Graphs are plotted for the following resolutions at distances of 5 meter, 10 meter and 15 meter:

Resolution	Number of bytes
160x120	88832
320x240	341504
640x480	1338368

Table 5.1 Resolution size

The frame numbers on X-Axis are actually an image which constitutes several numbers of packets as defined in following table. If a single packet is lost the complete image is discarded.

Resolution	Packet Size (bytes) x Number of Packets	Total Size (bytes)
160x120	8192 x 11	~88832
320x240	32768 x 11	~ 341504
640x480	65535 x 21	~ 1338368

Table 5.2 Packet size

5.1.1 Single Client

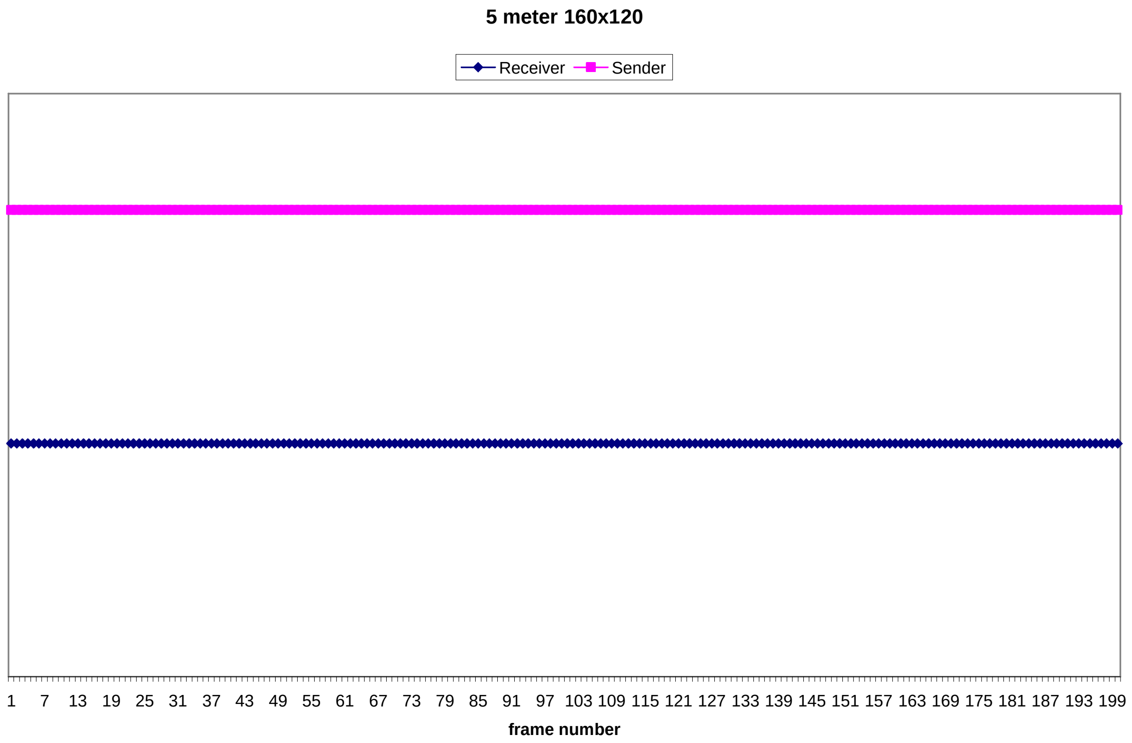

Figure 5.1 Single Client 5 meter Resolution 160x120

Time Interval: 100ms

Sender: 200 Frames

Receiver: 200 Frames

Loss: 0

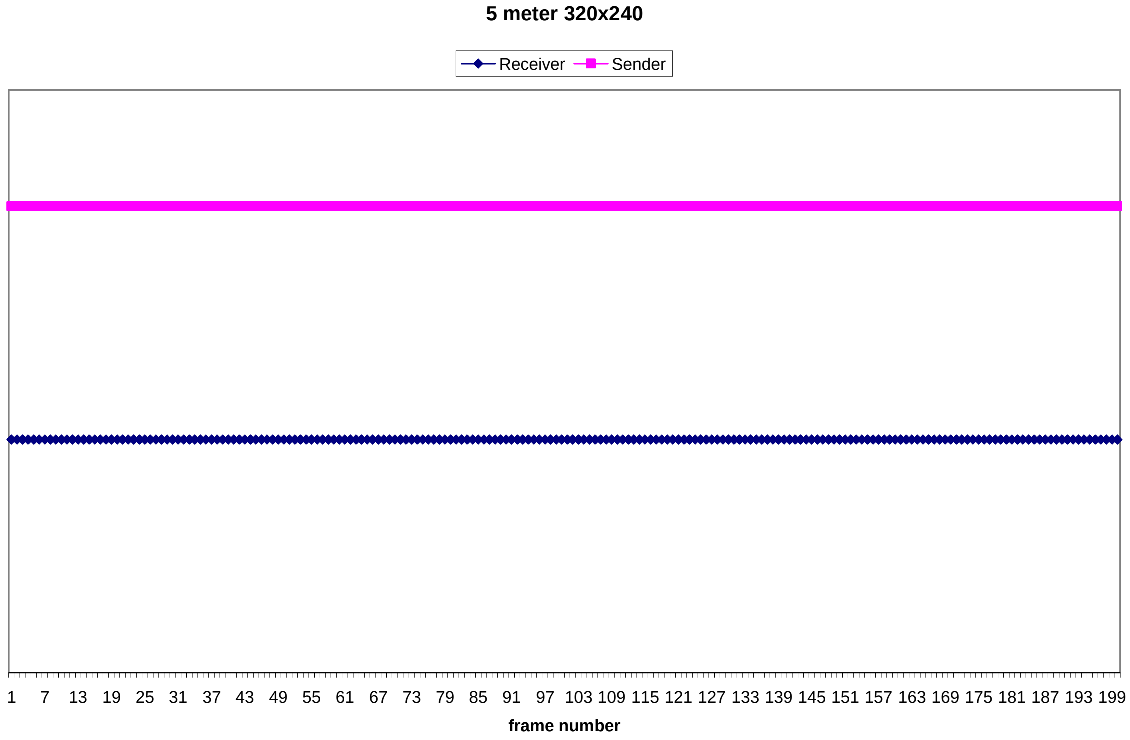

Figure 5.2 Single Client 5 meter Resolution 320x240

Time Interval: 200ms

Sender: 200 Frames

Receiver: 200 Frames

Loss: 0

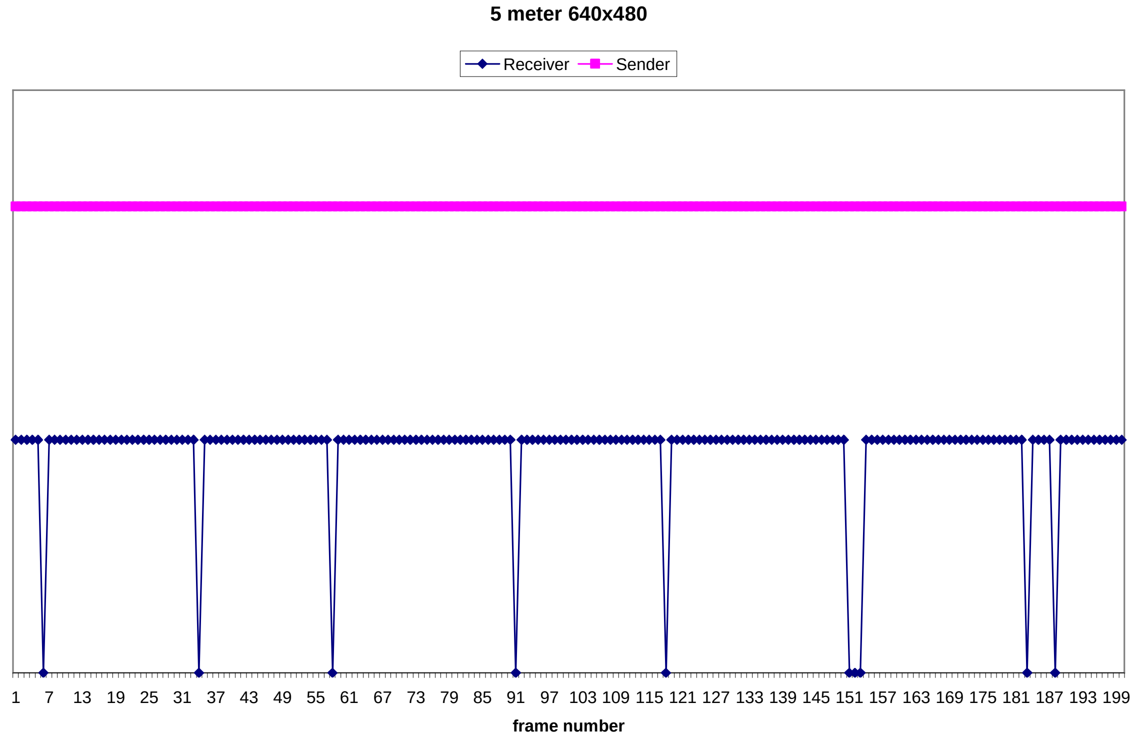

Figure 5.3 Single Client 5 meter Resolution 640x480

Time Interval: 800ms

Sender: 200 Frames

Receiver: 191 Frames

Loss: 9 frames

Loss Rate = 4.5%

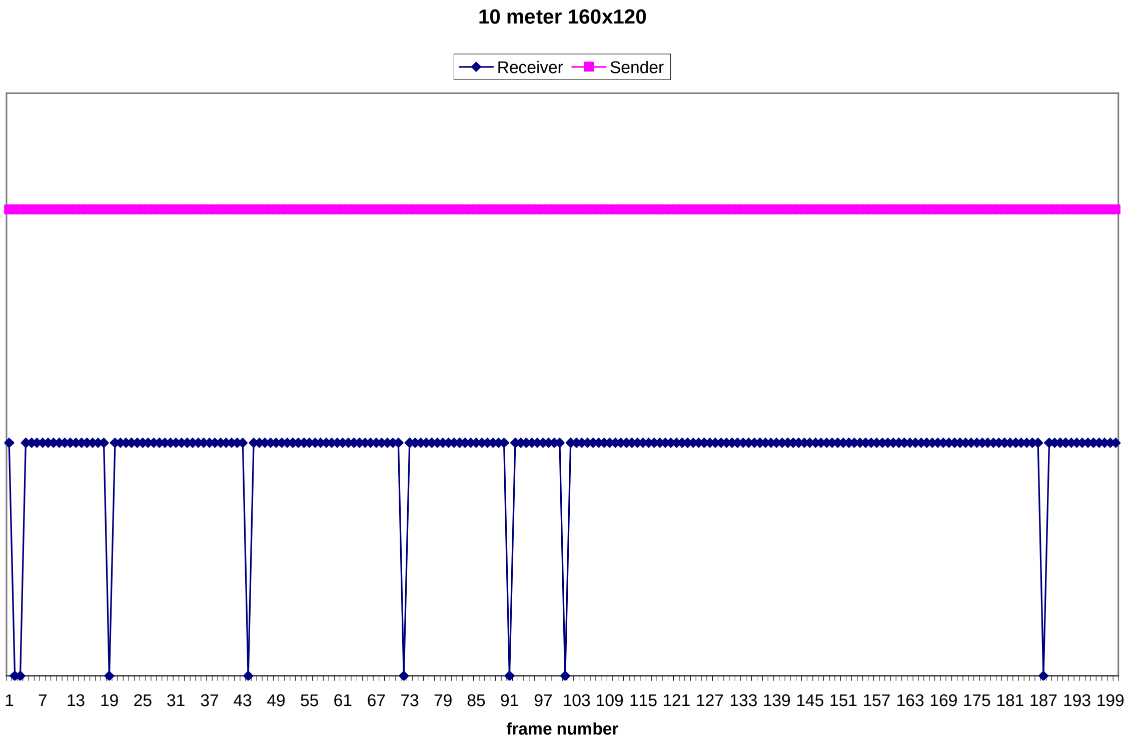

Figure 5.4 Single Client 10 meter Resolution 160x120

Time Interval: 100ms

Sender: 200 Frames

Receiver: 192 Frames

Loss: 8

Loss Rate = 4%

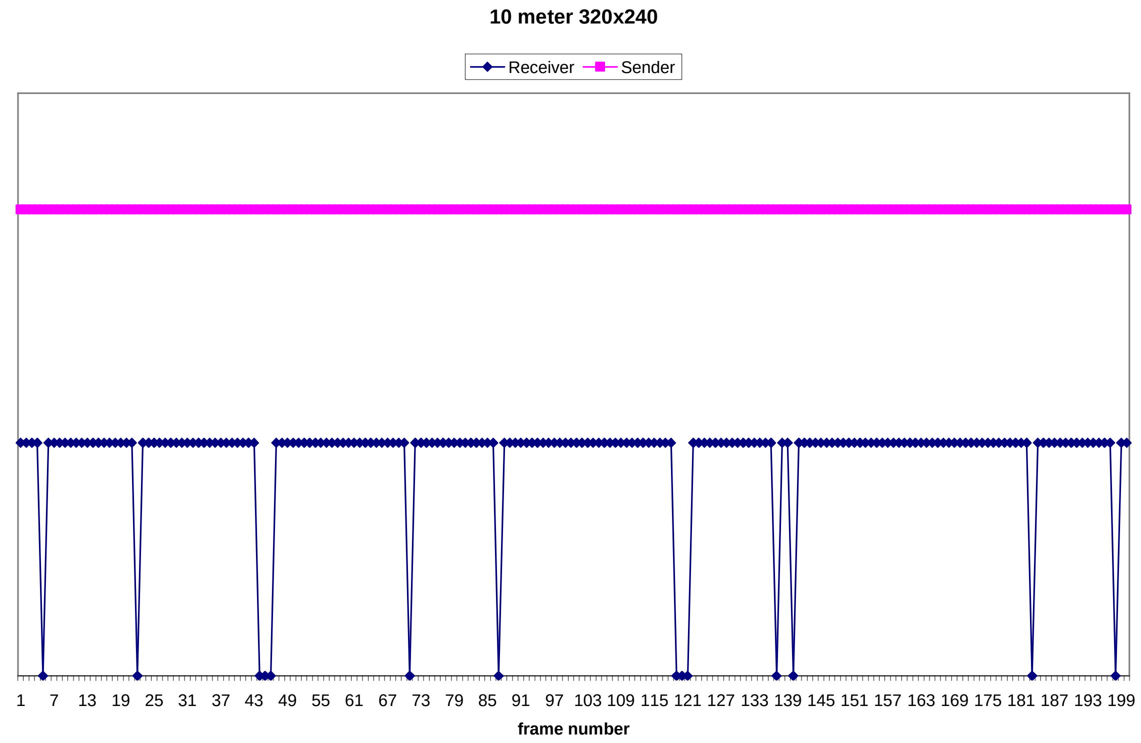

Figure 5.5 Single Client 10 meter Resolution 320x240

Time Interval: 200ms

Sender: 200 Frames

Receiver: 188 Frames

Loss: 12

Loss Rate = 6%

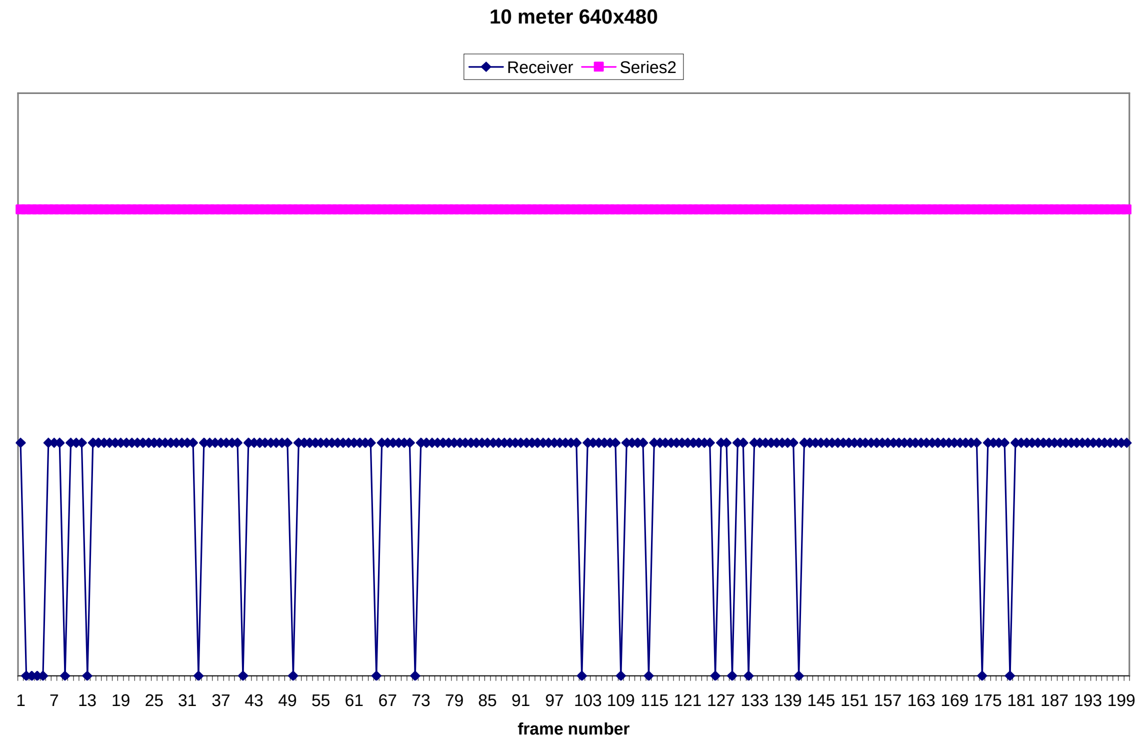

Figure 5.6 Single Client 10 meter Resolution 640x480

Time Interval: 800ms

Sender: 200 Frames

Receiver: 181 Frames

Loss: 19

Loss Rate = 9.5%

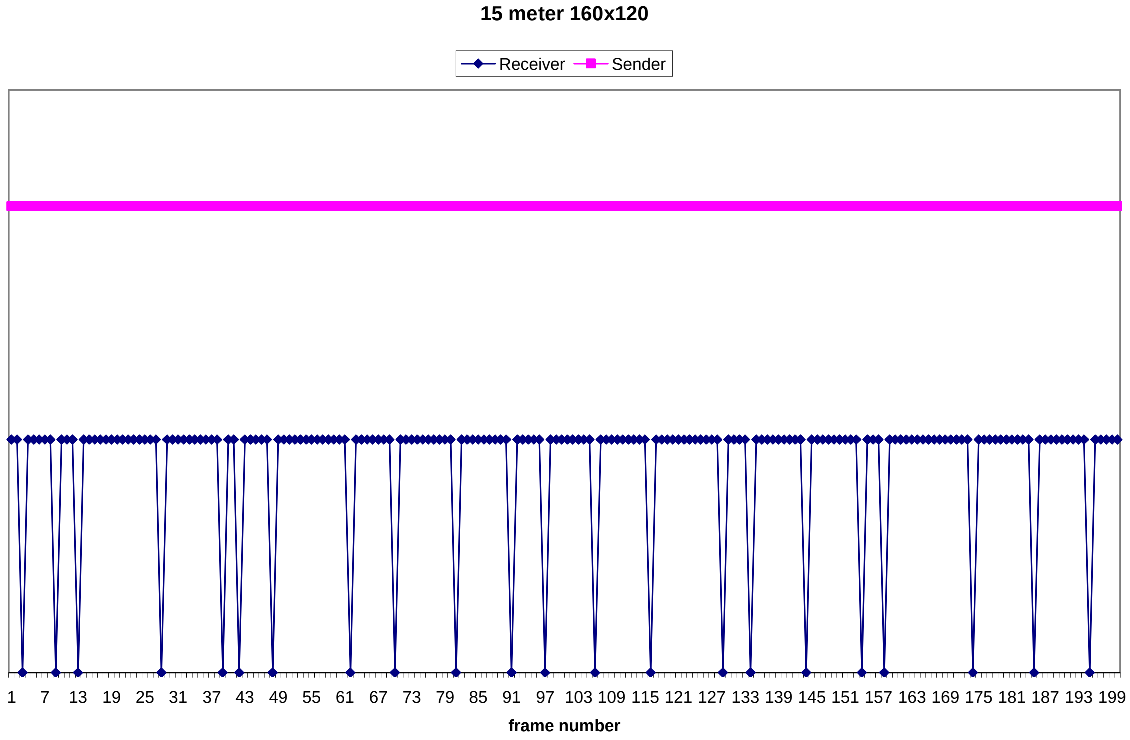

Figure 5.7 Single Client 15 meter Resolution 160x120

Time Interval: 200ms

Sender: 200 Frames

Receiver: 178 Frames

Loss: 22

Loss Rate = 11%

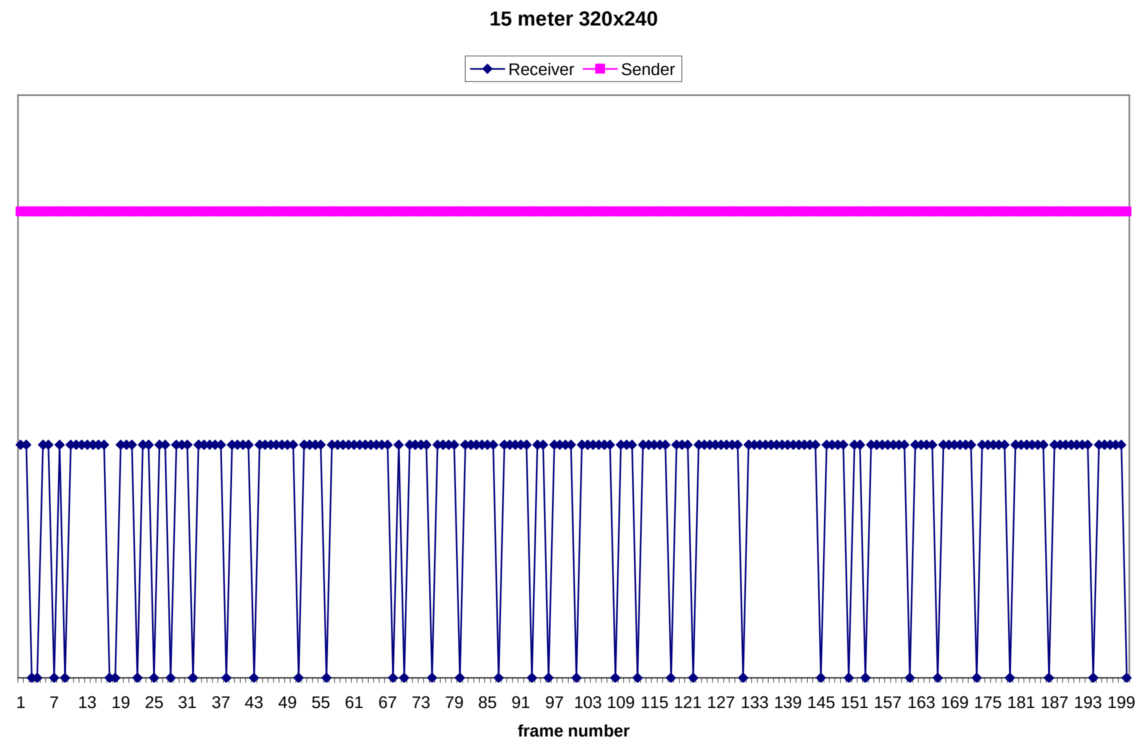

Figure 5.8 Single Client 15 meter Resolution 320x240

Time Interval: 400ms

Sender: 200 Frames

Receiver: 163 Frames

Loss: 37

Loss Rate = 18.5%

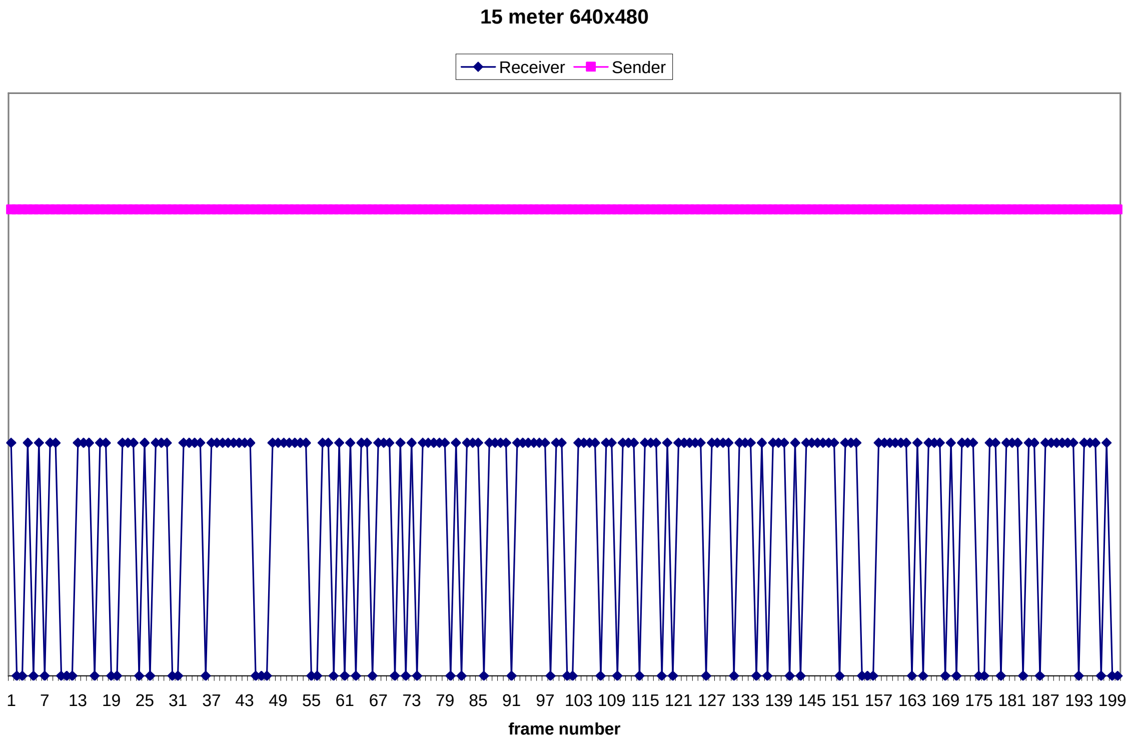

Figure 5.9 Single Client 15 meter Resolution 640x480

Time Interval: 2000ms

Sender: 200 Frames

Receiver: 137 Frames

Loss: 63

Loss Rate = 31.5%

5.1.2 Multiple Client

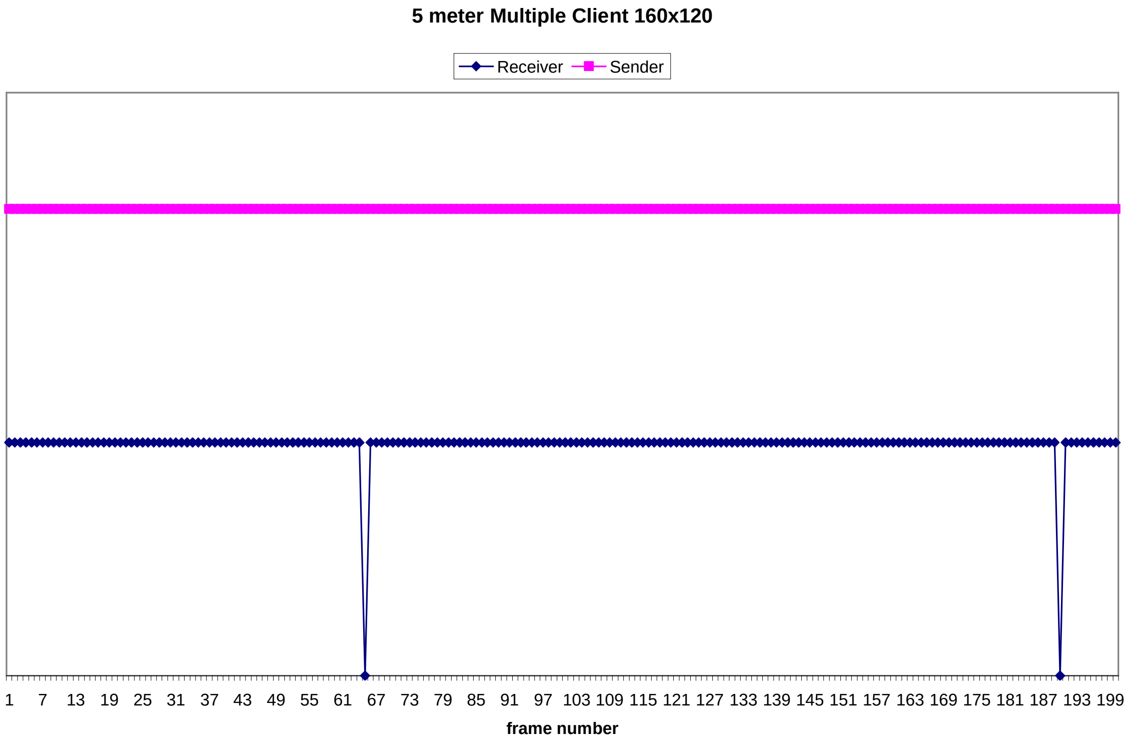

Figure 5.10 Multiple Client 5 meter Resolution 160x120

Time Interval: 100ms

Sender: 200 Frames

Receiver: 198 Frames

Loss: 2

Loss Rate = 2%

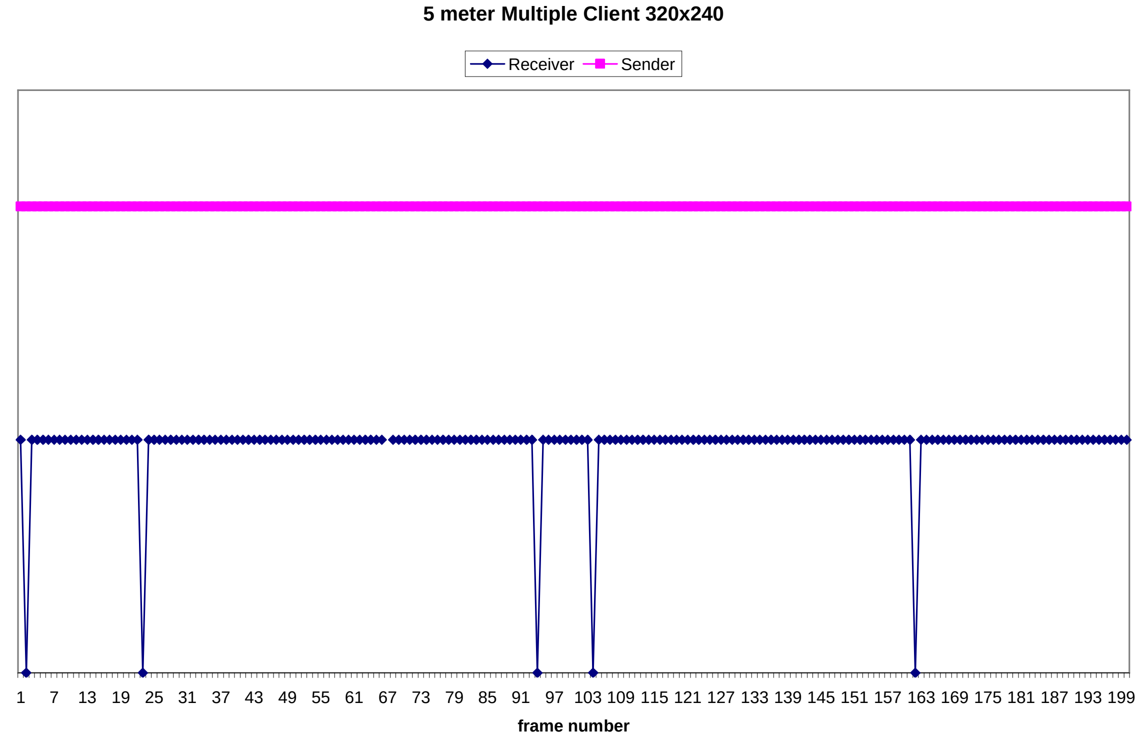

Figure 5.11 Multiple Client 5 meter Resolution 320x240

Time Interval: 200ms

Sender: 200 Frames

Receiver: 195 Frames

Loss: 5

Loss Rate = 2.5%

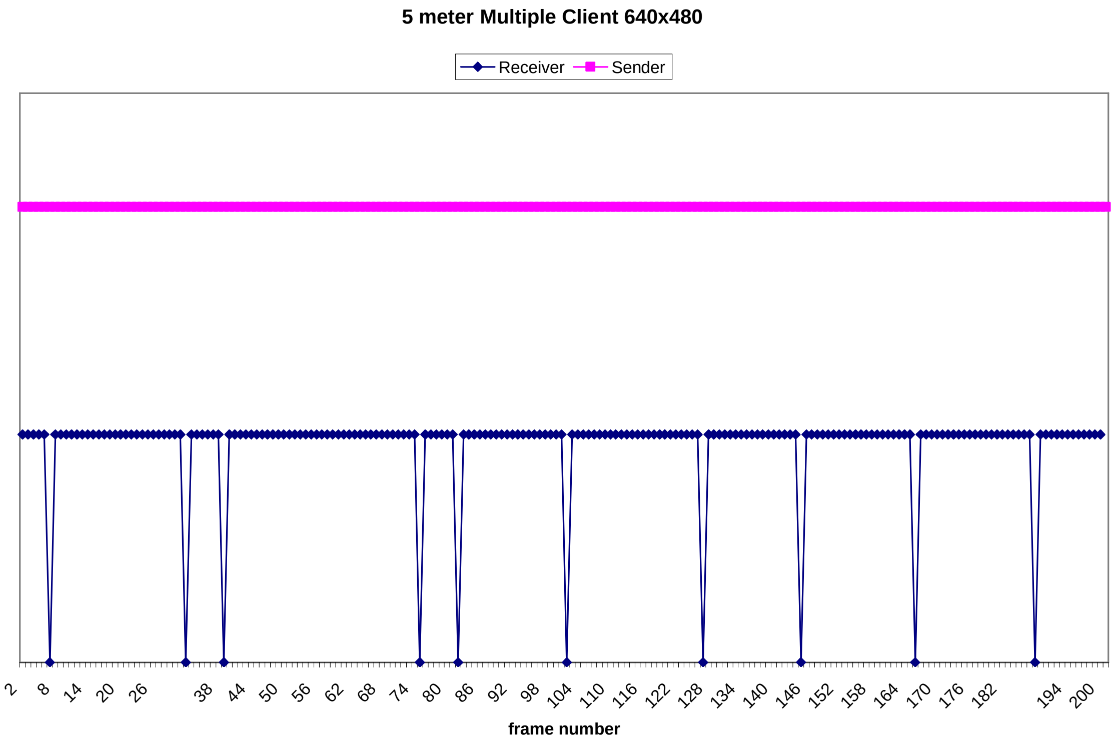

Figure 5.12 Multiple Client 5 meter Resolution 640x480

Time Interval: 800ms

Sender: 200 Frames

Receiver: 190 Frames

Loss: 10

Loss Rate = 5%

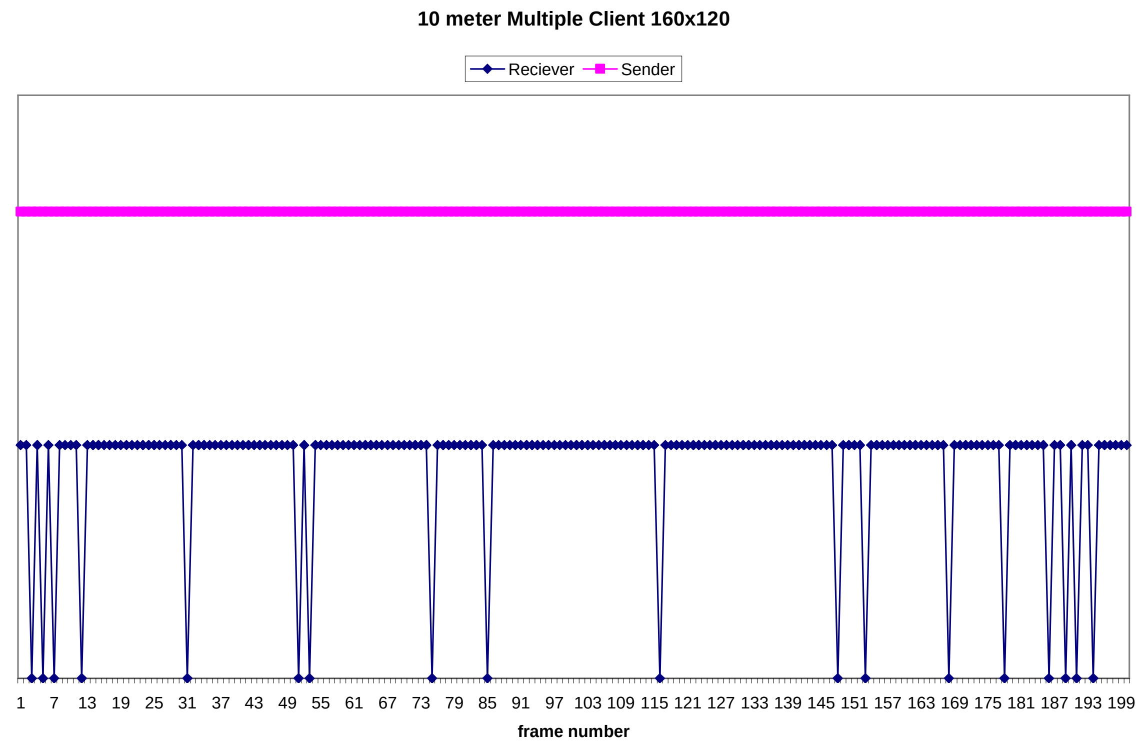

Figure 5.13 Multiple Client 10 meter Resolution 160x120

Time Interval: 200ms

Sender: 200 Frames

Receiver: 192 Frames

Loss: 18

Loss Rate = 9%

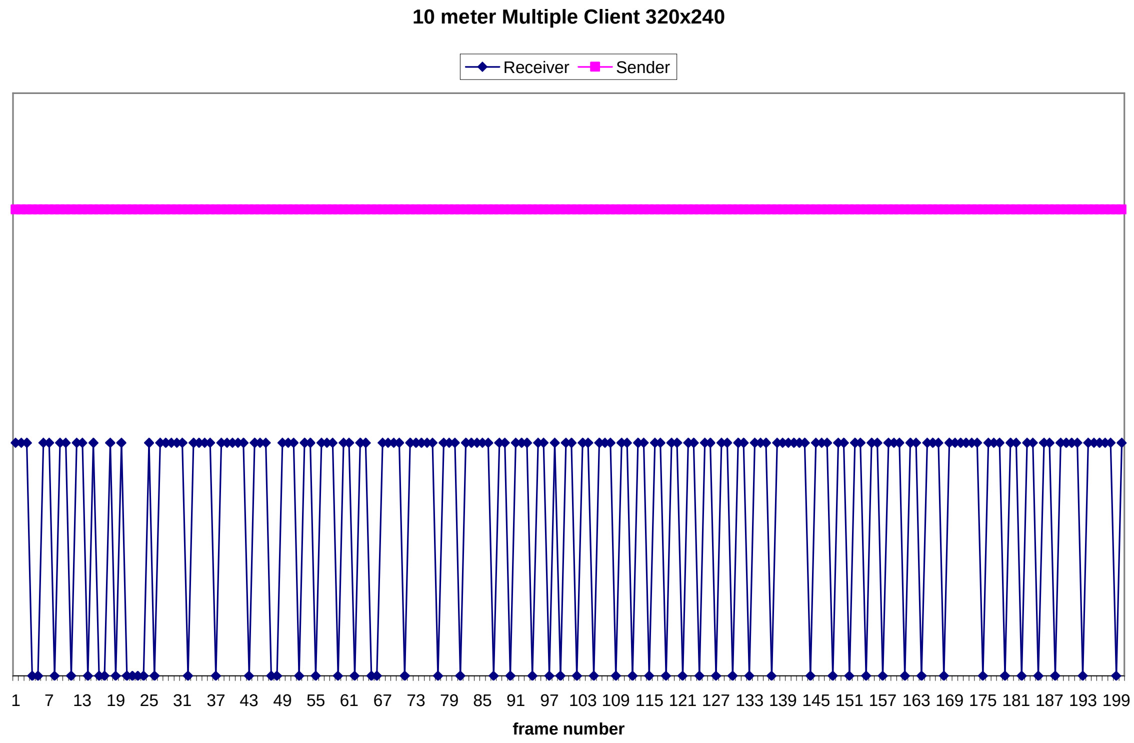

Figure 5.14 Multiple Client 10 meter Resolution 320x240

Time Interval: 400ms

Sender: 200 Frames

Receiver: 140 Frames

Loss: 60

Loss Rate = 30%

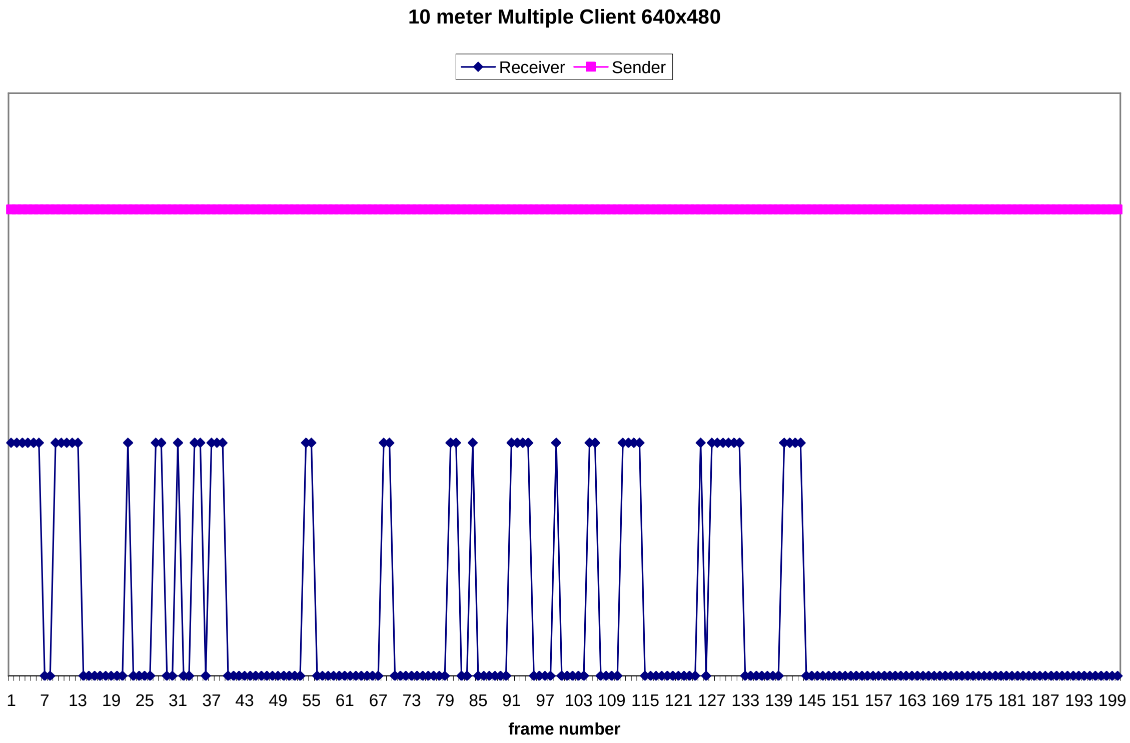

Figure 5.15 Multiple Client 10 meter Resolution 640x480

Time Interval: 2000ms

Sender: 200 Frames

Receiver: 110 Frames

Loss: 90

Loss Rate = 45%

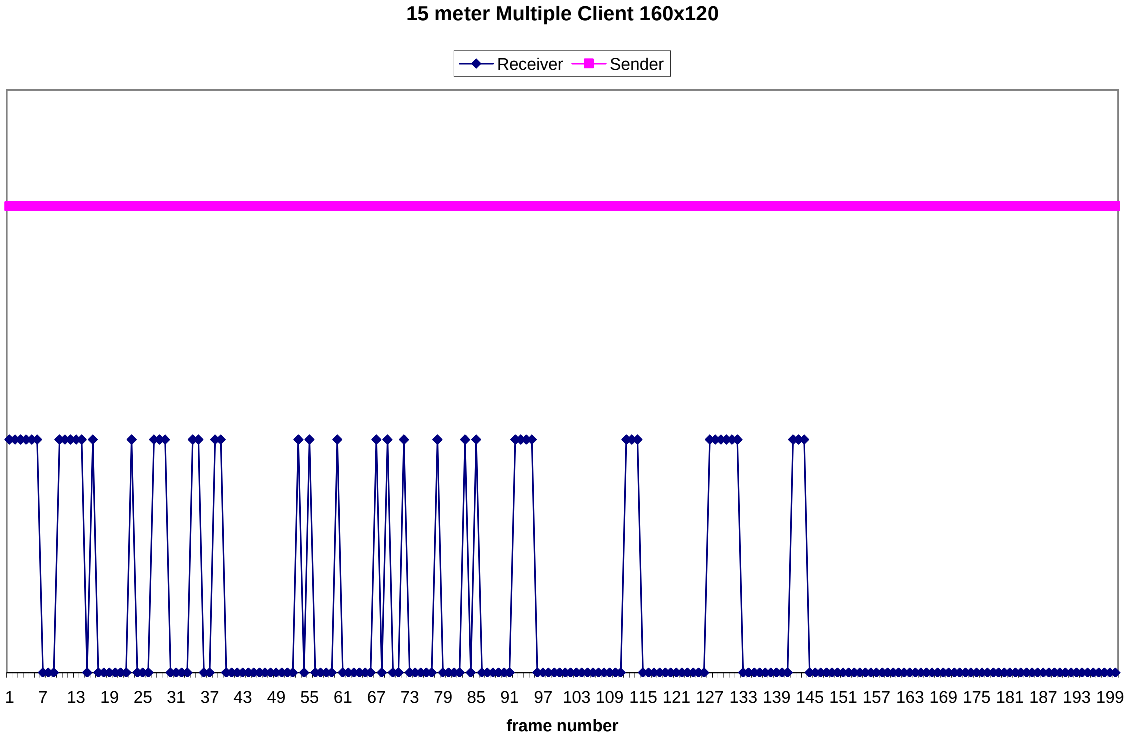

Figure 5.16 Multiple Client 15 meter Resolution 160x120

Time Interval: 800ms

Sender: 200 Frames

Receiver: 98 Frames

Loss: 102

Loss Rate = 51%

Increasing time interval to 1200ms will also give same loss rate of 51.5%, further increasing time interval doesn't helps.

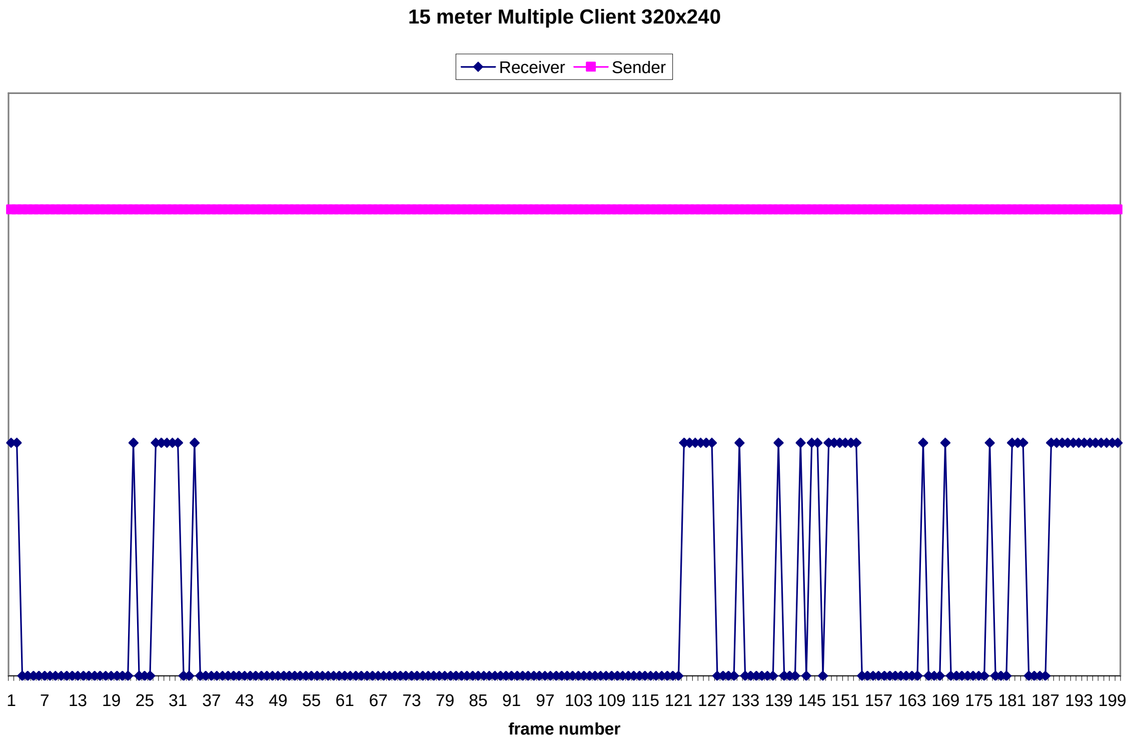

Figure 5.17 Multiple Client 15 meter Resolution 320x240

Time Interval : 2000ms

Sender: 200 Frames

Receiver: 45 Frames

Loss: 155

Loss Rate = 77.5%

If time interval is 800ms then screen hangs after showing 12 frames for a long time and then again displays 4 frames before complete stop. If Time interval is increased further to 3000ms receiver gets 48 frames.

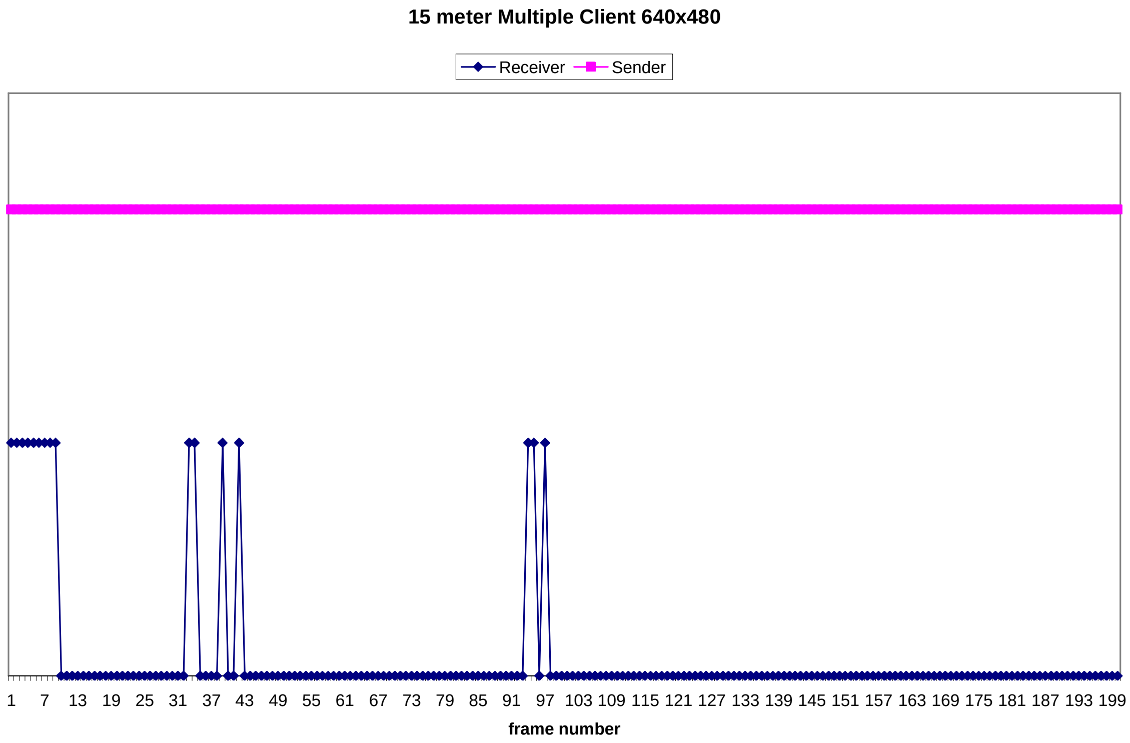

Figure 5.18 Multiple Client 15 meter Resolution 640x480

Time Interval: 4000ms

Sender: 200 Frames

Receiver: 17 Frames

Loss: 183

Loss Rate = 91.5%

Frame number 97 is the last frame received by receiver. If time interval is 2000ms then screen hangs after showing 4 frames. Increasing time interval further to 5000ms showed 16 frames before screen froze to frame number 92.

Now we will see the case in which we will reduce the packet size to 8192 bytes. So total of 164 packets needs to be sent.

i.e. 8192 x 164 ~ 1338368 for 640x480 resolution.

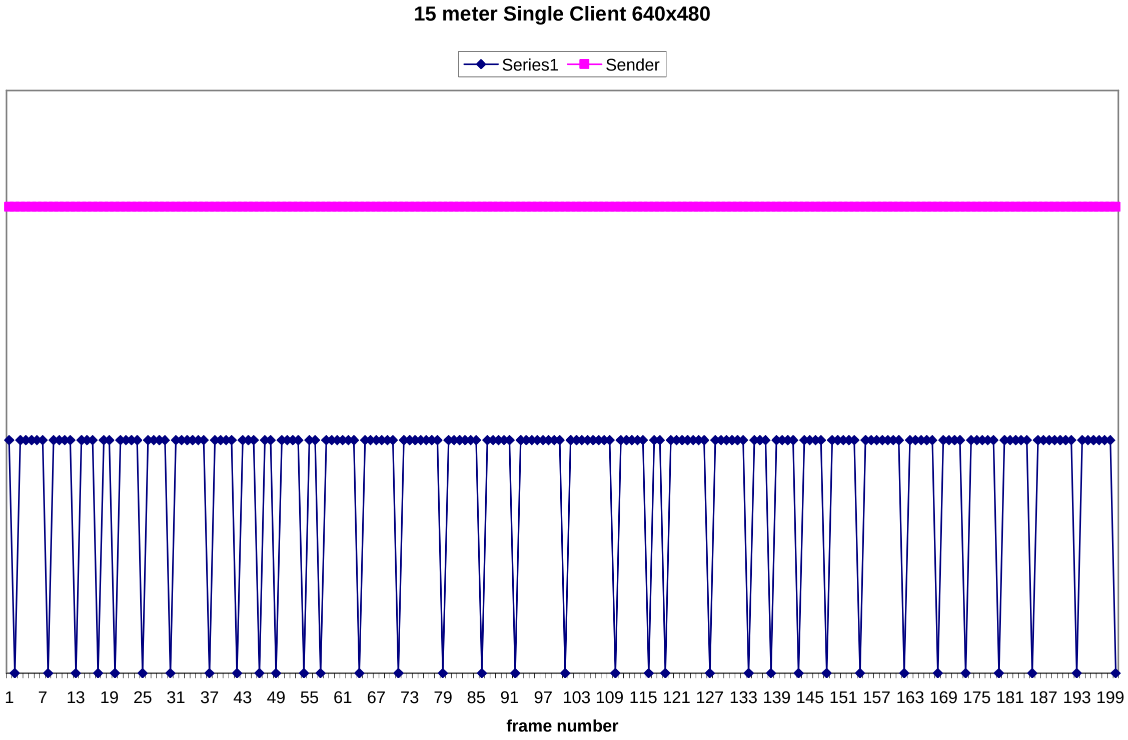

Figure 5.19 Single Client 15 meter Resolution 640x480 Reduced Packet Size

Sender: 200 Frames

Receiver: 165 Frames

Loss: 35

Loss Rate = 17.5%

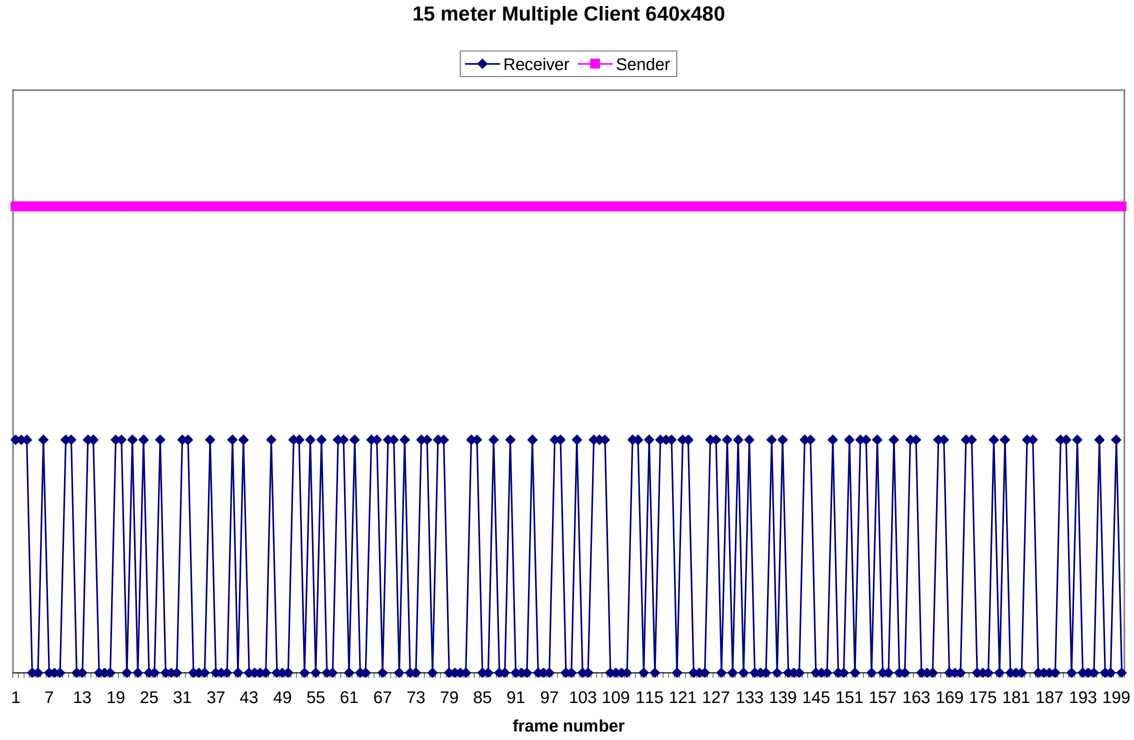

Figure 5.20 Multiple Client 15 meter Resolution 640x480 Reduced Packet Size

Sender: 200 Frames

Receiver: 82 Frames

Loss: 118

Loss Rate = 59%

Reducing the packet size of each frame have increased the number of packets for the frame but it gives better results as compared to bigger packet sizes. For single client 15 meter range 640x480 resolution is giving better performance as compared to the one with bigger packet size same is the case with 15 meter Multiple Client for 640x480 resolution.

This reason to this can be:

- Machines' input buffer fills up so packets that arrive while the buffer is still full are dropped.

- If we send larger packets, IP layer fragments it further and sends those packets which might be sent very quickly and thus result in packet loss. But when we send packets of smaller size which are already fragmented, the packets are sent at a more even rate which may result in less loss.

We will now study the time difference between each frame which arrives at client with respect distance between server and client for all the resolutions and compare the results. Following graph represents the time difference between each frame received at client end. As UDP doesn’t guarantees delivery of each datagram sent in the network as there is no retransmission in UDP. The crest in a graph represents the shoot up time when any frame gets lost. Also we can see that the rate at which sender sends data is different from the rate at which receiver receives the data this is because receiver in this case is iPAQ which has low processing power than a computer machine. So the time difference considerably increases for very large data like size of frame for resolution 640x480 is 1338368 bytes which considerably increases the time to download a frame and thus video appears to be slow, it even slows down further if distance between server and client is increased.

5.1.3 Single Client Ad hoc network

1) Resolution 160x120

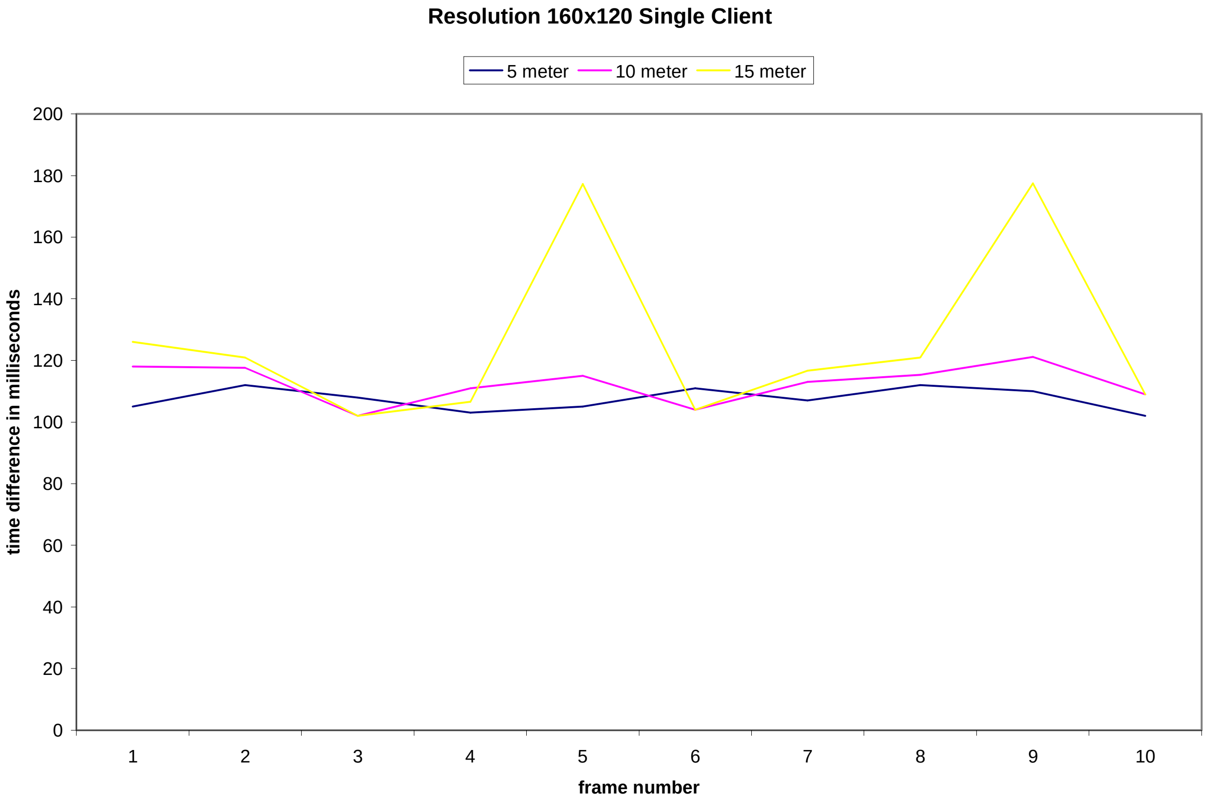

Figure 5.21 Resolution 160x120 Single Client

For the resolution 160x120 we can see that 5 meter and 10 meter range readings give almost straight line representing constant time difference between each frame received around 110 – 120 milliseconds which is almost same as the rate at which server sent.

This is not the case with 15 meter range as two sharp crests observed for 15 meter range are the datagram packet lost during transmission which suddenly increased the time difference between frame 4 and frame 5 as the frame in between them was lost. Similarly in all the following graphs crests represents packet loss.

2) Resolution 320x240

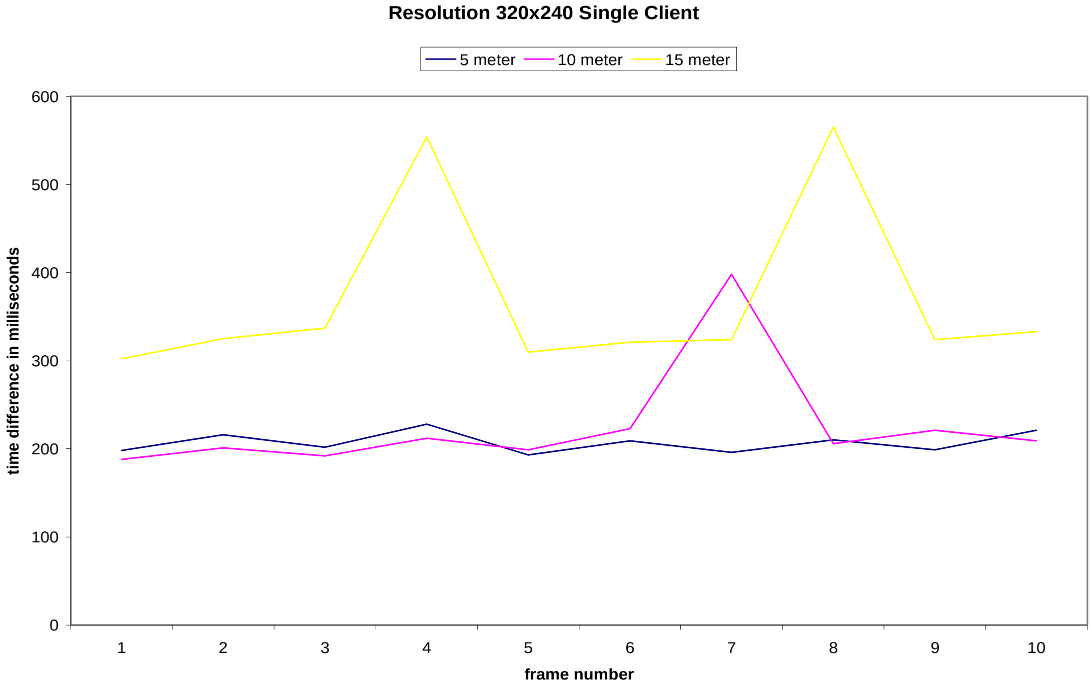

Figure 5.22 Resolution 320x240 Single Client

3) Resolution 640x480

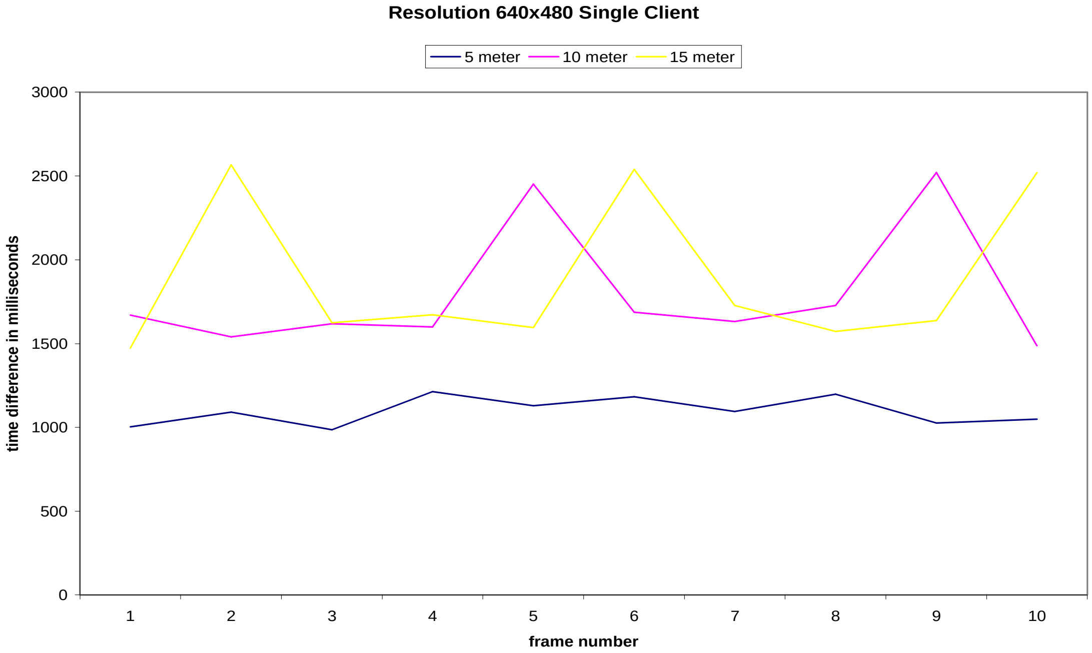

Figure 5.23 Resolution 640x480 Single Client

Y-Axis of graph represents the time each frame since last frame arrived at client while X-Axis is the frame number. For 5-meter and 10 meter range system is performing consistently and as the distance is increased to 15 meter we can see there are couple of steep lines which represents extra time receiver has to wait for next frame. This is because some frame(s) gets lost in transmission thus increasing the receiving time of next frame. The transmission protocol used is UDP and we may have packet loss in this protocol. So it clearly justifies the loss of frames.

Also we can observe that for lower resolution and closer distance value of time component is significantly less which makes streaming of video appropriate at lower resolution and closer distance in ad hoc network and as the resolution increases the size of data to transfer increases thus time taken also increases and the speed of streaming video is slow.

5.1.2 Multiple Client Ad hoc Network

1) Resolution 160x120

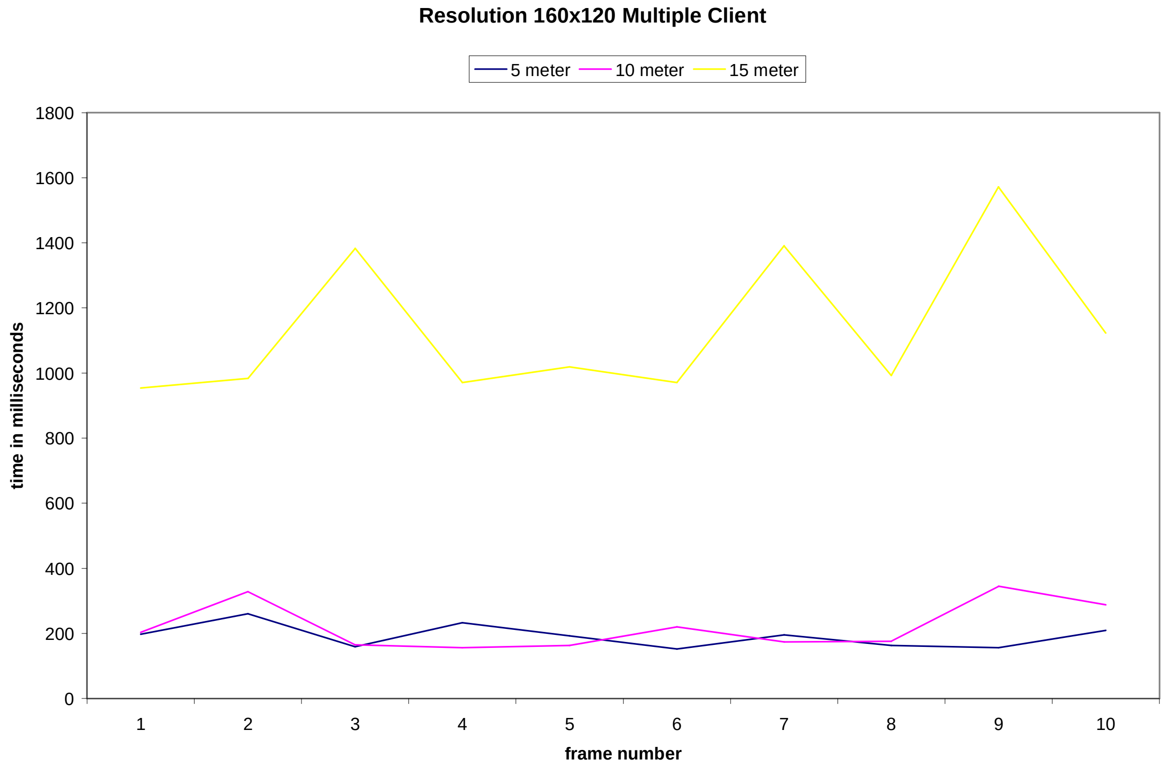

Figure 5.24 Resolution 160x120 Multiple Client

2) Resolution 320x240

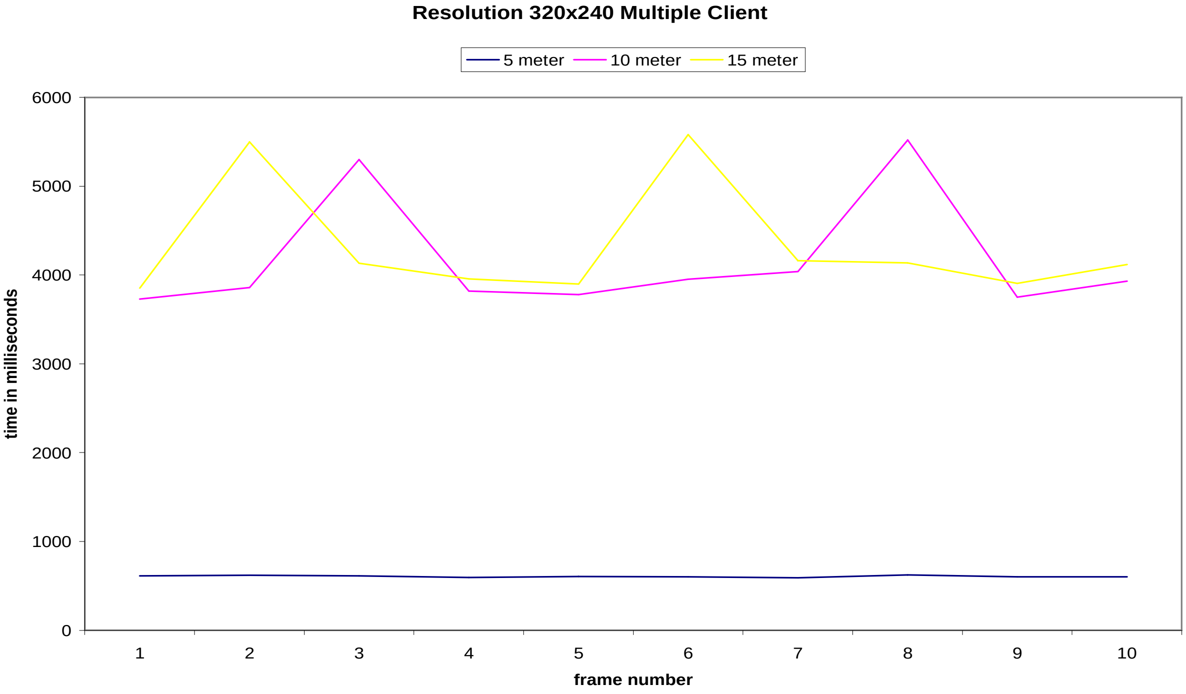

Figure 5.25 Resolution 320x240 Multiple Client

3) Resolution 640x480

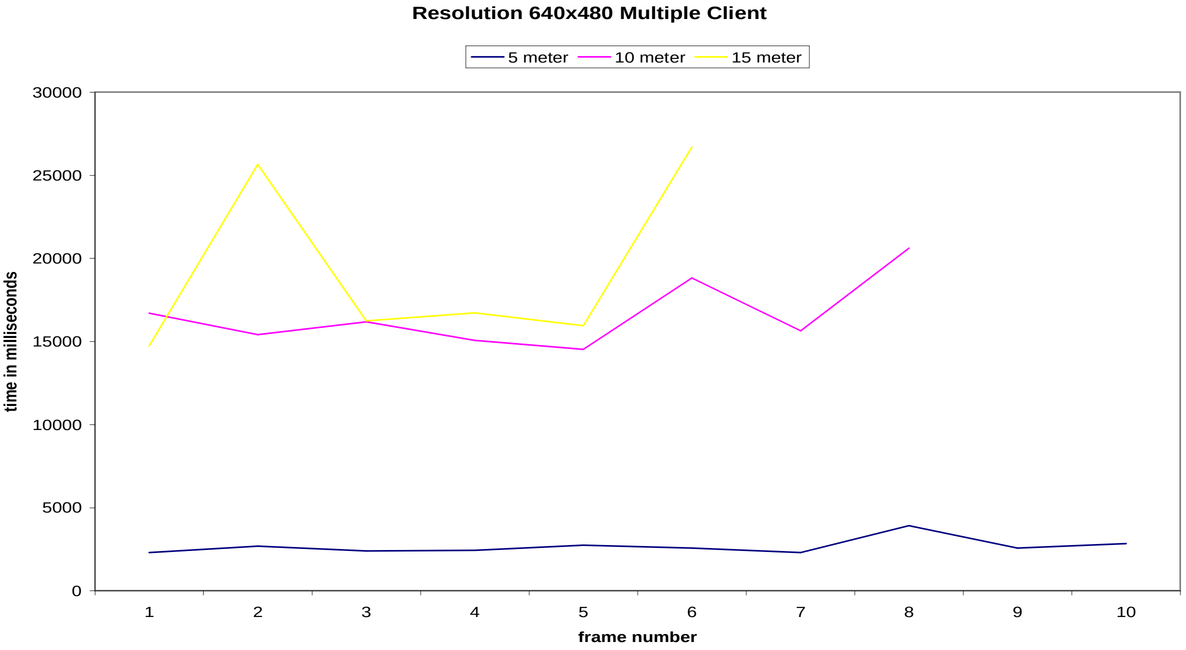

Figure 5.26 Resolution 640x480 Multiple Client

For multiple clients we can see more packet loss as compared to single client especially for 10 meter and 15 meter range. As network traffic increases with introduction of more clients the chances of packet getting lost increases as explained above. In the case of Resolution 640x480 for 15 meter and 10 meter range the connection with server broke after couple of frames during experiment and there were no frames further received by client. As the data is large for this resolution and with multiple clients connected to server client lost connection to server.

5.2 Bottlenecks

In a client-server architecture over an ad hoc network with low capability computation devices like iPAQs there can be some bottlenecks which may result to system halt. Unlike conventional networks ad hoc network do not suffer from bandwidth limitations so that criteria is discarded. But ad hoc wireless network depends on total number of nodes in the network and distance between the two nodes sending and receiving data wirelessly. These two criterions can be a bottleneck at some point of time when number of nodes participating i.e. number of clients in the network exceeds the limit which an ad hoc network can support also when the distance between the client and server is more than the communication range. Also as the client in this case is a handheld device which has low computation power than a pc can be a bottleneck to itself as it may not able to receive images of very high resolution. Also the input buffers of a handheld device are smaller than a pc which may again prove to be a bottleneck for large image sizes. Handheld devices have low CPU capabilities and small memory as compared to a computer machine and receiving video streaming requires large buffer space to store some of the packets until all the packets are arrived. So for large resolution image low CPU computation power and memory can prove to be a bottleneck. At server side it has multiple threads like to listen for connection, handle connection request, stream video and video capture, some times shared variables can lead to bottlenecks by forming a deadlock or a livelock in which system comes to an halt and server can’t handle any more clients neither it can satisfy the request of existing clients. Such condition can be a bottleneck for system and leads to undesired behavior of the complete system. Synchronization locks are used to prevent system from such kind of behavior.

CHAPTER 6 - Conclusion

From the analysis of the result it is clear that proper balance needs to be achieved between resolution and distance of client and server in ad doc network. For lower resolution and closer distance system is giving good performance by showing real time picture of what is happening in the field but increasing resolution of video will decrease the speed of video as data to transfer increases also for resolution value 160x120 the performance is system is consistent for all distances 5 meter, 10-meter, 15 meter. But as we increase resolution of video the performance of system degrades and streaming of video gets slow down. Balanced video streaming in terms of resolution and distance for handheld devices over ad hoc network gives appreciable results considering the fact that handheld devices have low processing power than a computer machine.
This project can be further enhanced to be used in Sensor Networks to sense video data .

References

[1] Video Streaming: Concepts, Algorithms and Systems

http://www.hpl.hp.com/techreports/2002/HPL-2002-260.pdf

[2] Transmission Control Protocol & User Datagram Protocol

http://msdn.microsoft.com/en-us/library/ms810606.aspx

[4] Microsoft Specific classes

http://msdn.microsoft.com/enus/library/system.net.sockets.tcpclient.getstream(VS.71).aspx

[5] Streaming Media

http://en.wikipedia.org/wiki/Streaming_media