國 立 交 通 大 學

電信工程學系

碩 士 論 文

配備頻寬平順技術之 RTP/RTSP 即時互動式多媒體串流監控系統設計與實作

Design and Implementation of a Real-time Interactive RTP/RTSP Multimedia Streaming Monitoring System with Bandwidth Smoothing Technique

研究生 ： 陳建華

指導教授： 張文鐘 博士

中 華 民 國 九十四 年 八 月

配備頻寬平順技術之
RTP/RTSP 即時互動式多媒體串流監控系統
設計與實作

Design and Implementation of a Real-time Interactive RTP/RTSP Multimedia Streaming Monitoring System with Bandwidth Smoothing Technique

研 究 生：陳建華	Student：Chien-hua James Chen
指導教授：張文鐘 博士	Advisor：Dr. Wen-Thong Chang

國 立 交 通 大 學
電 信 工 程 學 系
碩 士 論 文

A Thesis
Submitted to Department of Communication Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Communication Engineering

August 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年八月

配備頻寬平順技術之
RTP/RTSP 即時互動式多媒體串流監控系統
設計與實作

研究生: 陳建華　　　　　　　　　　指導教授: 張文鐘 博士

國立交通大學電信工程學系碩士班

摘要

多媒體串流（Streaming）技術的目的，是使消費者在下載多媒體檔案的同時，可以先觀賞已收到的部份，而不需要等到完整下載後，才開始觀賞。目前最被廣泛接受的串流通訊協定是 Real Time Protocol(RTP)，它是針對即時串流的特殊需求所設計的。此協定也可搭配 Real Time Streaming Protocol (RTSP)使用。RTSP 提供 Server 與 Client 之間的雙向溝通，Client 可以透過 RTSP 下指令給 Server 作像是「播放」、「暫停」、「停止」等動作。市面上有許多軟體廠已有解決方案，此些 Servers 都符合 RTP 及 RTSP 的規範，但只提供單純的多媒體檔案串流。此些 Servers 無法支援如即時串流無線攝影機所或 USB 攝影機所拍攝到的影像，或將變動位元速率 (Variable-bit-rate) 的多媒體檔案，做規劃平整 (Smoothing) 的動作，來達到較平順的傳送等。

本論文針對此需求，提出一套符合 RTP 及 RTSP 且可以很輕易被擴充功能的 Client/Server 軟體系統平台 。在本系統裡，Server 端可以向各種攝影機擷取影像及聲音，即時的壓縮到 MPEG-4 的格式，儲存或馬上串流到多個 Clients。Client 可向 Server 要求串流攝影機的即時影像和聲音，或已儲存的多媒體檔案等。Client 如果要求的是已儲存的檔案，Server 會先針對 Video 的統計特性，做頻寬平整的動作，達到較佳的串流效果及網路資源的利用。Client 若為監控中心，因為此系統提供簡單的軟體輸出/輸入界面，所以可在系統上輕鬆的加裝影像處理程式，達到自動化監控的效果。本論文將仔細地介紹此系統的架構、設計想法及使用方法，尤其會特別闡述如何使 Server 可以在非常準確的時間，將每張 frame 傳送出去。

本文所提出的目標及期望達到的功能都已達到和實現。然而，在觀察 Server 的傳送情況後發現，每張 frame 被傳送出去的時間準確度並不理想，且此誤差在多個 clients 連線時將更明顯。本文在最後將提出可改善此系統的方向及未來可努力的目標。

Design and Implementation of a Real-time Interactive RTP/RTSP Multimedia Streaming Monitoring System with Bandwidth Smoothing Technique

Student: Chien-hua James Chen   Advisor: Dr. Wen-Thong Chang

Department of Communication Engineering
National Chiao Tung University

Abstract

Streaming technology allows people to enjoy the multimedia contents while still downloading. Up to date, the most widely used and accepted streaming protocol is Real Time Protocol (RTP), which is specially designed for the needs of real-time streaming, and can be used in conjunction with Real Time Streaming Protocol (RTSP). RTSP provides bidirectional communication between the server and the client. Several software companies have come out with their streaming solutions that comply with RTP and RTSP. However, they merely fulfill the needs of streaming stored media files. They can neither stream real-time captured video/audio acquired from various types of cameras, nor transmit VBR multimedia contents in a smoother manner.

In this work, an RTP/RTSP-compliant client/server streaming system that is flexible enough to be added with new functions is implemented. In particular, the server can acquire audiovisual data from different kinds of cameras, real-time encoding them into MPEG-4 format, and store or stream the resulted bitstreams to multiple clients. The client can ask the server to stream live-captured or stored media contents. If the client requests the stored video, the server can run the smoothing algorithm to smoothen the VBR traffic, so a better streaming experience and more efficient utilization of network resources can be obtained. Since the client software is designed to provide a clean and easy-to-use software input/output interface, image processing functions can be added to it effortlessly to allow automotive monitoring of unattended areas. Certainly, the architectures, design ideas, and the usages of the client and server systems will be explained in details. How the server can stream data following tight timing constraints will be addressed specifically.

Apparently, the proposed goals are achieved and expected functions are realized. However, it is observed at the server side that the inter-departure times between frames are not equal to the expected values. This discrepancy becomes more obvious when more clients are present. Finally, possible ways to enhance the proposed system and the directions of the future work will be suggested.

Acknowledgements

I would like to express my deepest gratitude to my academic and research advisor Dr. Wen-Thong Chang for his guidance and constant support in helping me conduct and complete this work. I would also like to thank my colleagues in Wireless Multimedia Communication Laboratory for their active participation in this research. Many thanks to all the people I have come to know in National Chiao Tung University, whose friendship and companionship I will always enjoy.

I owe my sincere appreciation to my families and relatives who have supported and encouraged me over the years. I especially want to thank Dephnie Li for her inspiration and continuous encouragement during my studies. Most important of all, I want to extend my profound appreciation to my beloved parents and families, for their love, affection, and invaluable support during my life and studies.

Table of Contents

Lists of Figures

List of Tables

Acronyms

API	Application program interface
CBR	Constant-bit-rate
IETF	Internet Engineering Task Force
MTU	Maximum transmission unit
MJPEG	Motion JPEG
MPEG	Motion Picture Experts Group
MVBA	Minimum Variance Bandwidth Allocation
RCBS	Rate-Constrained Bandwidth Smoothing
RTCP	The RTP Control Protocol [part of RFC3550]
RTP	The Real-time Transport Protocol [RFC3550]
RTSP	The Real Time Streaming Protocol [RFC2326]
URI	Uniform Resource Identifier
VBR	Variable-bit-rate

1 Introduction

1.1 Background

With wide spread of network infrastructure and extensive usage of both wired and wireless devices, consumers can enjoy multimedia contents on their PCs, set-top boxes, Digital TVs, and mobile phones at any time and any place.   For example, the user can use his or her PC to download the latest episode of *Star Wars* from one of the content providers. When the downloading process is completed, the user can start watching the movie. This is extremely convenient because the user does not need to go out when he or she feels lazy or it is raining outside. The user is only one “press” away from enjoying the high resolution media contents. This way of retrieving and enjoying the media contents is referred as the traditional downloading method.

So far, the story sounds terrific and seems to end perfectly. However, in fact, the story is not even close to what it looks like. For instance, what if one is in the coffee shop and suddenly wants to watch a movie of two-hour long? Can he or she wait there for the completion of downloading? Two-hour move is usually of size greater than 1 GB, and if only IEEE 802.11b (11Mbps) wireless network is available in the coffee shop, he or she will have to wait for approximately 10 to 20 minutes if the network is not congested. If the network is shared by multiple users (congested), which is often the case, he or she will need to wait a lot longer. For another example, what if one is using a handheld device (e.g. a PDA or mobile phone) that has much lower transmission rate and much lesser memory space (around 256 to 512 MBs)? He or she will have to wait for couple of hours to download the complete movie. In addition, the worst thing is that the memory is not even sufficient to store the complete media file!

To overcome this serious problem, the technology called “streaming” has been developed. Instead of transmitting the whole file, the content server will divide the media file into tiny minipackets (usually around 1024 bytes), and transmit them to the user according to the transmission schedule specified for the file. When the player that the user is using receives the stream of packets, it will start to render them on the display device. Thus, the streaming technology allows one to start watching or listening to the media content as soon as he or she has received a certain amount of the content. Of course, the content provider has to continuously transmit the remaining data with the transmission rate equal to or higher than the playback rate to guarantee smooth (no delay) presentation at his or her display device. The major distinction between “streaming” and the traditional downloading is that the former offers “play while downloading” while the later merely provide “play after downloading”. In short, streaming enables playback of audiovisual content in real time, while with the traditional downloading method, the entire file has to be retrieved before playback begins.

Actually, media streaming has emerged as an important service offered over the Internet in these days. There are two types of streaming: pseudo streaming and true streaming [1]. The former replies on the TCP/IP, the Internet protocol used for conventional web pages, while the later employs, up to date, the most widely used and accepted streaming protocol, Real Time Protocol (RTP) [2], which is defined in RFC-3550 by Internet Engineering Task Force (IETF),and is specially designed for the needs of real-time streaming. According to [2], RTP provides end-to-end delivery services including payload type identification, sequence numbering, timestamping, and transmission monitoring, for multimedia data with real-time properties, such as video and audio. Typically, RTP does not work on its own but do rely on and run on top of suitable underlying network such as UDP (primarily) or TCP. Moreover, RTP consists of two major components:

- The real-time transport protocol (RTP), which carries real-time data.
- The RTP control protocol (RTCP), which monitors reception quality and exchanges information about the participants in an active session.

RTP can be used in conjunction with Real Time Streaming Protocol (RTSP) [3], which is defined in RFC-2326 by IETF and layered on top of RTP. RTSP acts as a “network remote control” to provide bidirectional communication between the server and the client similar to HTTP. According to [3], RTSP establishes and controls one or more than one time-synchronized stream of media data. For example, in a client/server scenario with RTSP functionalities, the client can issue “PLAY”, “PAUSE, and “STOP” requests to control the streaming process of the server.

Since the streaming technology has been realized on RTP and enhanced by RTSP, and high-speed network infrastructure has been widely spread, one may think of the application that for some places like wards in hospitals or rooms with unattended sick elders (targeted areas or places) that need to be automatically monitored because of limited budget to hire caretakers or a shortage of nurses, several wireless cameras can be installed to help monitor the status of these targeted areas. These cameras will send their captured audiovisual contents to a centralized streaming server in real time. The client system with automotive image processing functions can ask the server to stream these live captured contents to it, so it can analyze the contents. When discovering abnormal events, the client system may notify the related personnel by sending him or her an SMS or email. The related personnel, upon receiving the notification, will connect to the streaming server immediately to receive the stored or the most recently recorded audiovisual contents, and perform the necessary actions. To realize this application, at first thought, the system designer may want to use one of the free streaming servers available in the market. In fact, at present several software companies have come out with their streaming solutions, such as the Media Server and Player from Microsoft, the Helix Server and Player of Real, and the Darwin Streaming Server and QuickTime Player of Apple. These solutions do comply with RTP and RTSP, but however, they only fulfill the need of streaming stored multimedia contents. They neither can work with various kinds of cameras, nor stream live-captured audiovisual contents. In addition, they do not provide automotive image processing functions for surveillance purpose.

Furthermore, to enhance the streaming experience, the system designer may want to control the actual streaming process. For example, the system designer can adapt the transmission rate to the congestion level of the network. In reality, due to the VBR nature of the common media contents, it is not easy to allocate the network resources for streaming. The burstiness of these contents does add burden to the network. Therefore, a streaming control technique called “smoothing” has been developed to overcome this disadvantage. Its task is to smoothen the VBR traffic so that the traffic will behave CBR-like. However, the aforementioned commercial solutions do not adopt this concept into their transmission control. Therefore, it is necessary to implement an RTP/RTSP-compliant client/server streaming system that can work with any type of camera, provide simple interface for image processing software developers to embed their programs into the client system, and offer the options of whether to turn on the smoothing function or not.

1.2 Motivations

The motivations of this work are as the following:

- Commercial streaming systems cannot support streaming of live-captured audiovisual contents acquired from various types of cameras.
- Image processing programs allow the surveillance system to monitor unattended targeted places automatically. However, the commercial streaming solutions do not offer this function.
- The VBR characteristic of media contents adds burden to the network, so their transmission schedules should be smoothened by smoothing algorithm. However, the commercial streaming solutions do not adopt this concept.

1.3 Research Goals

The goals of this research are as follows:

- Implementing a client/server system that is fully compliant with RTP and RTSP.
- Designing the system in such a way that it provides the options of whether to turn on the smoothing functions or not.
- Designing a simple software interface for the client system in such a way that image processing algorithms can be added to it in an effortless manner.
- Describing how to build, organize, and use this system.
- Discussing the execution of the system.

1.4 Thesis Outline

The organization of this paper is as follows. We survey the current available streaming servers developed by Apple, Real, and Microsoft in Chapter 2. In Chapter 3, RTP and RTSP are introduced and discussed, and some programming techniques needed to implement the system are mentioned and explained. The implementation details, usage of the system, and execution results are presented in Chapter 4. At last, this project is concluded in Chapter 5.

2 Streaming Technologies

In this section of the paper, streaming technologies involved to design and implement the proposed client/server streaming system will be discussed. In particular, the Real-time Transport Protocol (RTP) [2], its subpart, the RTP Control Protocol (RTCP) [2], and Real Time Streaming Protocol (RTSP) [3] will be addressed in detail.

2.1 The Real-time Transport Protocol

In a client/server media streaming system, if video/audio data are transmitted using The Transmission Control Protocol (TCP) from a server to a client, intolerable delay will be introduced frequently in the case of packet losses. This drawback results from the fact that TCP always recovers these losses by retransmitting lost packets to provide reliable service, so the client must wait for receipt of all retransmissions. If this waiting time is significantly long, the client's playback buffer will soon be empty and the player will stop playback since it runs out of contents, causing extremely unpleasant viewing experiences. Thus, although TCP guarantees delivery of data but does not guarantee timely delivery as this retransmission mechanism causes additional end-to-end packet delay.

One of the solutions to solve this problem is to use a more light-weight but unreliable protocol, The Datagram Protocol (UDP). UDP, unlike TCP, does not support automatic retransmission, and cannot be used to detect or signal packet losses. Hence, UDP cannot adapt itself to the changing network conditions. However, in a congested network where packet losses do occur, it does provide a faster transmission than TCP.

The Real-Time Transport Protocol (RTP) [2] is a protocol which can compensate for the missing functions of UDP. It is usually used on top of UDP to make use of UDP's multiplexing and checksum services [2]. In fact, its packets are usually encapsulated in UDP packets. According to [2], RTP provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. The services provided include content type identification, sequence numbering, timestamping and monitoring QoS of data transmission. The detailed descriptions about RTP will be given in the following sections.

RTP consists of two tightly-linked protocols: RTP for transmission of data with real-time properties and RTP Control Protocol (RTCP) for monitoring QoS of transmission and for conveying participants' information in a session [2].

2.1.1 RTP

RTP provides end-to-end network transport functions suitable for applications transmitting real-time data, such as audio/video, over multicast or unicast network services. Its main purpose is to carry real-time audiovisual data. In other words, media data are packetized into several RTP packets in the format specified in the following section. These RTP packets are passed down to the lower layer protocol like UDP, which in turn, transmit them to the client side.

2.1.1.1 RTP Packet Format

As can be seen from Figure 1, RTP packets consist of a header followed by payload data which can be either a portion of video frame or audio samples. How different types of video/audio data are formatted to be carried by RTP packets is specified in payload format specification documents. For example, 3GPP MPEG4 AV payload format is defined in RFC3016 and Simple MPEG4 AV payload format is defined in RFC3640.

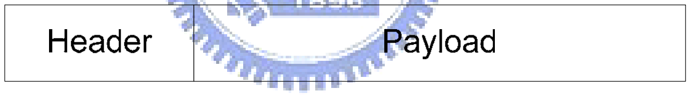


Figure 1 – RTP Packet Format

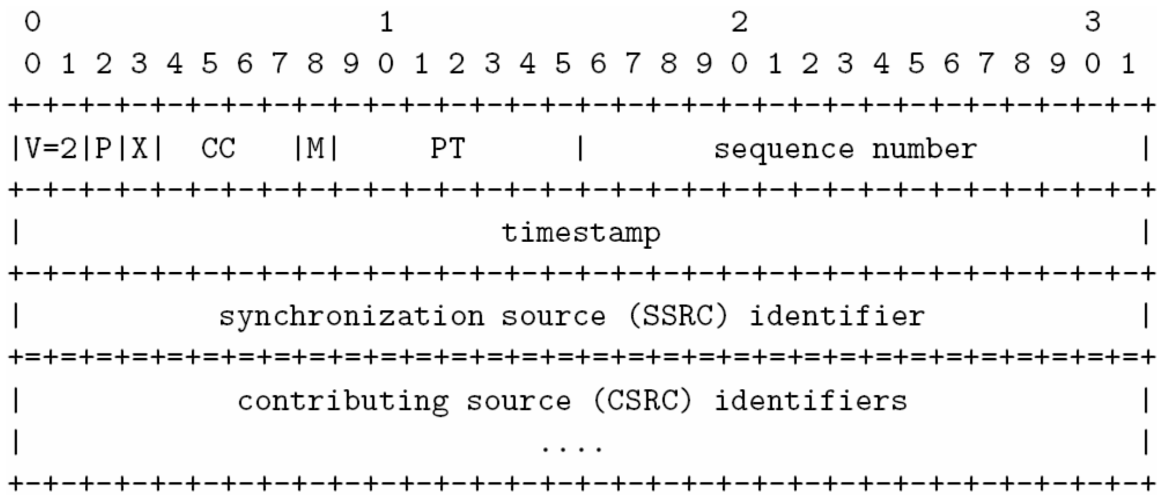
Referring to Figure 2, an RTP header has the following fields:

- **Version (V):** 2 bits

  It specifies the version of RTP. The Value 2 is set for RFC3550.

- **Padding (P):** 1 bit

  It indicates whether this packet contains padding bytes at the end. Padding may be necessary for some encryption algorithms or for carrying several RTP packets in a lower-layer data unit. The last byte of the padding stores the number of padding bytes that should be ignored including itself.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2 – RTP Header Format [2]

- **Extension (X):** 1 bit

  This bit is set when exactly one header extension is placed after the fixed header. The header extension is to allow individual implementations to conduct experiments.
- **CSRC count (CC):** 4 bits

  Contributing source (CSRC) is a source of a stream that contributes to the combined stream produced by a mixer. A mixer combines the packets from one or more sources in some manner, and forwards a new combined RTP packet. CC represents the number of CSRC sources of this RTP packet. The ideas of a mixer and the CSRC will become apparent in Section 3.1.2.2.
- **Marker (M):** 1 bit

  It is used to indicate the important events such as frame boundaries.
- **Payload type (PT):** 7 bits

  It identifies the format of the payload.
- **Sequence number:** 16 bits

  This number increments monotonically each time an RTP packet is sent. Its initial value should be random to avoid know-plaintext attacks on encrypted data. It can be used by the receiving end to detect packet loss or re-order out-of-sequence packets.
- **Timestamp:** 32 bits

  It reflects the instant when the first byte of the payload data is sampled for applications transmitting live-captured data, or when the payload data should be presented for applications transmitting stored data. As it reconstructs the timing of a single stream, it is used when performing synchronization or jitter calculation. Its initial value should be random to avoid know-plaintext attacks on encrypted data. Note that timestamps for different media streams may advance at different rates and usually have independent, random offsets.

- **SSRC:** 32 bits

  Synchronization source (SSRC) is the source of a stream of RTP packets. Its value is randomly selected to be globally unique within a particular RTP session. So if a single sender generates multiple streams in one RTP session, the possible confusion caused by only having a single network address can be avoided because the sender will have several distinct SSRC identifiers.

- **CSRC list:** 0 to 15 items, 32 bits each

  It is a list of CSRCs which contribute to the payload of this RTP packet.

2.1.1.2 RTP Example

To illustrate the notion of SSRC, CSRC, and a mixer more clearly, Figure 3 is presented. In the figure, participants A, B, and C, each having its own unique SSRC, are hosting a conference in an RTP session. Assume that the speed of the links between A and the mixer, and B and the mixer is high, while the bandwidth between the mixer and C is low. Obviously, C cannot enjoy high-speed network access like A or B. Instead of forcing all participants to transmit and receive poor-quality conference data, a mixer can be placed near C. In such arrangement, A and B can still enjoy their high-quality conference, because that the mixer, when receiving high-quality conference data from A and B, can translate, resynchronize, and combine these data into lower-bitrate ones, and forward them to C across the lower-speed link.

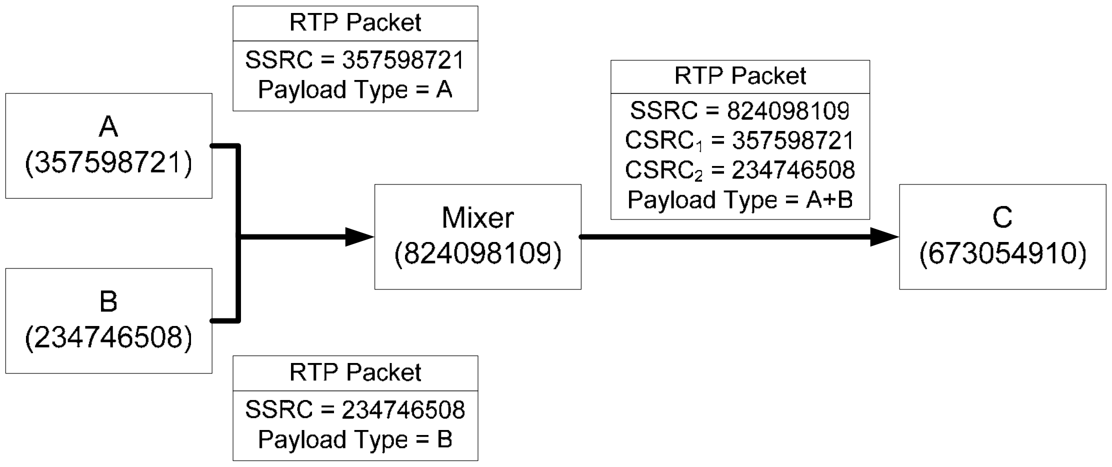


Figure 3 – An example of using SSRC, CSRC, and Mixer

2.1.2 RTCP

RTCP monitors the data delivery in a manner scalable to large multicast networks, and provides minimal control and identification functionality [2]. It is based on the periodic exchanges of control packets between all participants in an RTP session.

According to [2], RTCP provides the following three key functions:

- It gives feedbacks on the reception quality of a session by transmitting RTCP sender and receiver reports, as described in following section.
- It carries a canonical name (CNAME) for each participant. As the SSRC identifier of a participant may change if a conflict occurs or when a program is restarted, receivers need CNAME to keep track of each source.
- It can help the sender and the receiver adjust their clock drifts.

2.1.2.1 RTCP Packet Types

RTCP has five packet types:

- **SR:**

  - Sender report, for senders to transmit their transmission and reception statistics.
- **RR:**

  - Receiver report, for receivers to transmit their reception statistics, including number of lost packets, highest sequence number received, jitter, and other delay measurements to calculate the round-trip delay time.
- **SDES:**

- Source description items, including CNAME.

- **BYE:**

  - Indication of end of participation

- **APP:**

  -Application-specific functions

2.1.2.2 RTCP Sending Rules

To reduce the bandwidth consumption required by RTCP packets, multiple RTCP packets from a single participant can be concatenated to form a compound RTCP packet. Therefore, each RTCP packet must end on a 32-bit boundary to be stackable. There are two constraints that must be followed:

- Each compound RTCP packet must contain a report packet (SR or RR) to maximize the resolution of reception statistics, and this report packet must be the first to appear to facilitate header validation.
- Each compound RTCP packet must include SDES CNAME, so new receivers can identify sources of streams and begin to synchronize the media data streams as soon as possible. In another case, as the SSRC identifier of a participant may change if a conflict occurs or when a program is restarted, receivers need CNAME to keep track of each source.

In Figure 4, a typical compound RTCP packet is presented. Observe that at least one report packet and SDES CNAME are included.

RR	SDES:CNAME (phone:location)	BYE

Figure 4 – A typical compound RTCP packet

There are few general sending rules for RTCP packets [2]:

- The Traffic generated by RTCP should occupy 5% of an RTP session bandwidth.
- Sender reports should occupy one-forth of the RTCP bandwidth.
- The target interval between two RTCP packets must be larger than 5 seconds. However, the actual RTCP interval should be varied over the range between half and 1.5 times of the target interval.

2.1.2.3 RTCP Packet Formats

Sender Report

Sender reports are used to provide reception quality feedback and senders' information by participants who are senders, and possibly receivers at the same time. Referring to Figure 5, a sender report consists of three sections: the header, the sender info, and the report blocks. The fields in the header section are:

- **Version (V):** 2 bits

  - It specifies the version of RTP. The Value 2 is set for RFC3550.
- **Padding (P):** 1 bit

  - It indicates whether the compound RTCP packet contains padding bytes at the end of last individual RTCP packet. Padding may be necessary for some encryption algorithms.
- **Reception Report Count (RC):** 5 bits

  - It counts the number of reception report blocks contained in this compound RTCP packet.
- **Packet Type (PT):** 8 bits

  - Its value is set to 200 to identify this packet as an RTCP SR packet.
- **Length:** 16 bits

  - It represents the length of this RTCP packet including the header, the sender into, the report blocks, and any padding.
- **SSRC:** 32 bits

  - It indicates the synchronization source (SSRC) identifier for the originator of this sender report.

```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
header |V=2|P|    RC   |   PT=SR=200   |             length            |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                         SSRC of sender                        |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
sender |              NTP timestamp, most significant word             |
info   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |             NTP timestamp, least significant word             |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                         RTP timestamp                         |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                     sender's packet count                     |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                      sender's octet count                     |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
report |                 SSRC_1 (SSRC of first source)                 |
block  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  1    | fraction lost |       cumulative number of packets lost       |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |           extended highest sequence number received           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                      interarrival jitter                      |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                         last SR (LSR)                         |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                   delay since last SR (DLSR)                  |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
report |                 SSRC_2 (SSRC of second source)                |
block  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  2    :                               ...                             :
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
       |                  profile-specific extensions                  |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5 – Format of Receiver Report [2]

The sender information block contains the following fields.

- **NTP Timestamp:** 64 bits

  - It indicates the wallclock time in Network Time Protocol (NTP) format, which is in seconds relative to 00:00 UTC on January 1st, 1990, when this report was sent. The integral part is in the first 32 bits and the fractional part is in the last 32 bits.

- **RTP Timestamp:** 32 bits

  - It represents the same time instant as the NTP timestamp but is in the same units and random offset as the RTP timestamps in RTP packets. With RTP timestamps, intra-media and inter-media data can be synchronized for sources whose NTP timestamps are in synchronization.

- **Sender's Packet Count:** 32 bits

  - It contains the total number of RTP packets sent by the sender since the start of the transmission until now.

- **Sender's Octet Count:** 32 bits

  - It contains the total number of RTP data payload bytes sent by the sender since the start of the transmission until now.

The third section contains zero or more reception report blocks that have the following fields:

- **SSRC_n (source identifier):** 32 bits

  - It indicates the SSRC identifier of the source to which this reception report block is meant.

- **Fraction Lost:** 8 bits

  - It shows the percentage of RTP packets lost from SSRC_n since previous SR or RR was sent.

- **Cumulative Number of Packets Lost:** 24 bits

  - It contains the total number of RTP packets lost from SSRC_n since the start of the transmission until now.

- **Extended Highest Sequence Number Received:** 32 bits

  - The lower 16 bits represent the highest sequence number received from SSRC_n, and higher 16 bits show the number of sequence number wrap-ups.

- **Interarrival Jitter:** 32 bits

  - This field is an estimation of the statistical variance of RTP packet interarrival time, measured in timestamp units. Let S_i and R_i be the RTP timestamp and the time of arrival of i_{th} RTP packet, respectively. Let D(i,j) be the variation between the difference of i_{th} and j_{th} RTP packets' timestamps and the difference of these two packets' arrival times. Thus, D(i,j) can be computed as follows:

$$D(i,j) = (R_j - R_i) - (S_j - S_i) = (R_j - S_j) - (R_i - S_i)$$

The interarrival jitter should be calculated constantly as the following whenever an RTP packet arrives from SSRC_n:

$$J(i) = J(i-1) + (|D(i-1,i)| - J(i-1))/16$$

- **Last SR timestamp (LSR):** 32 bits

  - This field contains the center 32 bits out of 64 bits in the NTP timestamp included in the most recent SR from SSRC_n.

- **Delay since last SR (DLSR):** 32 bits

  - This field is the difference between the time the last SR was received and the time this reception report is generated. Referring to Figure 6, upon receiving this information, the sender can compute the round-trip propagation delay as RTT = (S_Clock – LSR) – DLSR, where S_Clock is the time this report is received.

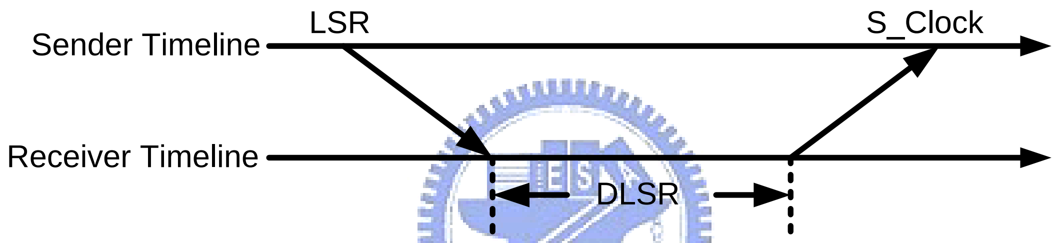
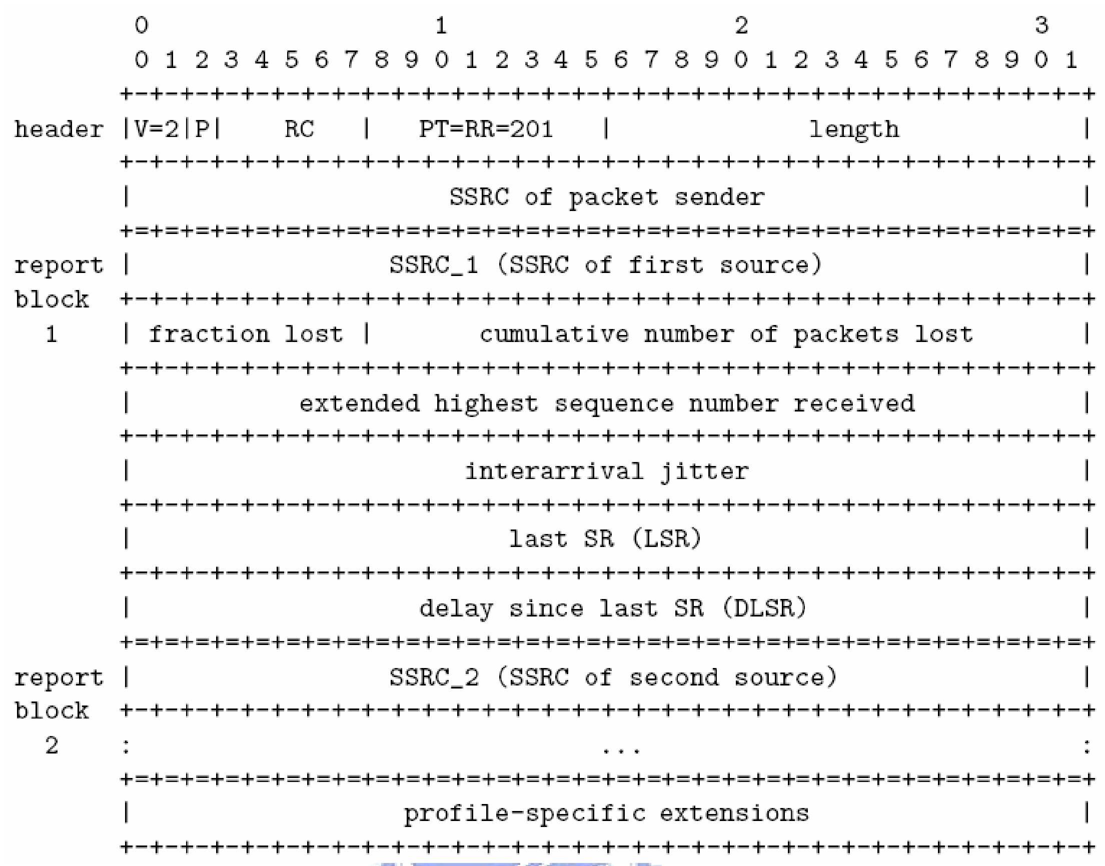


Figure 6 – Calculation of Round-trip delay

Receiver Report

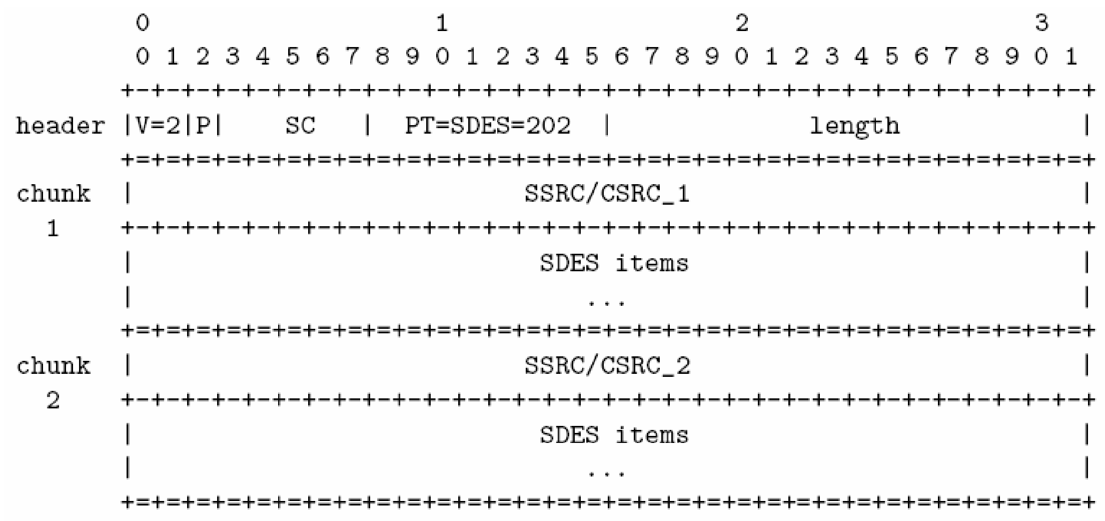
The purpose of the receiver report is identical to that of the sender report. Unlike the sender report, the receiver report is targeted for the receiver who is not also a sender at the same time. As can be observed from comparing Figure 7 with Figure 5, a receiver report is almost identical to a sender report except that the packet type field in a receiver report is 201, and that a receiver report does not contain the sender information. Therefore, the receiver report's format will not be explained in detail here.



```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
header |V=2|P|    RC   |   PT=RR=201   |             length            |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                     SSRC of packet sender                     |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
report |                 SSRC_1 (SSRC of first source)                 |
block  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  1    | fraction lost |       cumulative number of packets lost       |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |           extended highest sequence number received           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                      interarrival jitter                      |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                         last SR (LSR)                         |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                   delay since last SR (DLSR)                  |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
report |                 SSRC_2 (SSRC of second source)                |
block  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  2    :                               ...                             :
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
       |                  profile-specific extensions                  |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7 – Format of Receiver Report [2]

SDES



```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
header |V=2|P|    SC   |  PT=SDES=202  |             length            |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
chunk  |                          SSRC/CSRC_1                          |
  1    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                           SDES items                          |
       |                              ...                              |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
chunk  |                          SSRC/CSRC_2                          |
  2    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                           SDES items                          |
       |                              ...                              |
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
```

Figure 8 - Format of SDES [2]

Referring to Figure 8, an SDES packet consists of a header and zero or more chunks, each of which is composed of items describing the source that this chunk is designated to. In the header section of a SDES packet, following fields are present:

- **Version (V), Padding (P), Length:**
  - They are the same as those defined in the format of SR packet.
- **Packet Type (PT):** 8 bits
  - Its value is set to 202 to identify this as an SDES packet.
- **Source count (SC):** 5 bits
  - It is the number of SSRC/CSRC chunks contained in this SDES packet.

There are eight types of items but only CNAME is mandatory. Each type of item is composed of the type, the length, and the text.

■ **CNAME:** Canonical End-Point Identifier SDES Item

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    CNAME=1    |     length    | user and domain name        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9 – Format of CNAME item [2]

CNAME item should be formatted as in Figure 9. CNAME is used to eliminate the ambiguity caused when having SSRC conflicts. It must be included in each compound RTCP packet to provide the binding between SSRC and CNAME. Therefore, it should be unique among all participants within a session, remain unchanged during the lifetime of a session, and be derived algorithmically. For example, a CNAME can be “james@nctu.edu.tw” or “james@140.113.13.81”, where the part before ‘@’ is the user name and the part after ‘@’ is the host name.

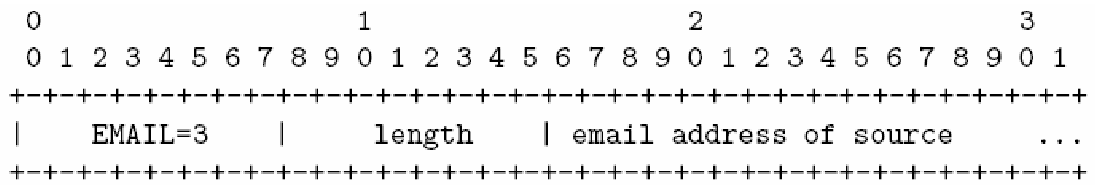
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     NAME=2    |     length    | common name of source       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 10 – Format of NAME item [2]

■ **NAME:** User Name SDES Item

NAME item should be formatted as in Figure 10. It contains the real name of the source. For example, it can be “James Chen”.

■ **EMAIL:** Electronic Mail Address SDES Item

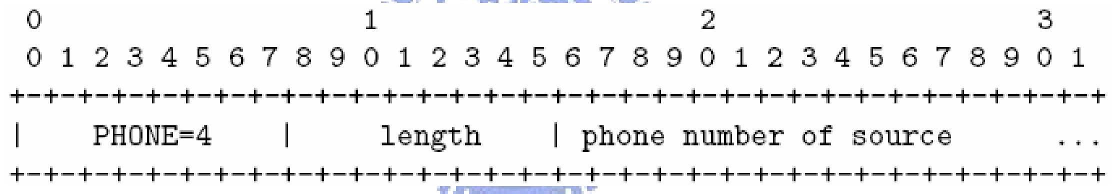


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    EMAIL=3    |     length    | email address of source     ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 11 – Format of EMAIL item [2]

EMAIL item should be formatted as in Figure 11. It contains the email address of the source and is expected to remain fixed for the lifetime of a session. For example, it can be “james@yahoo.com”.

■ **PHONE:** Phone Number SDES Item

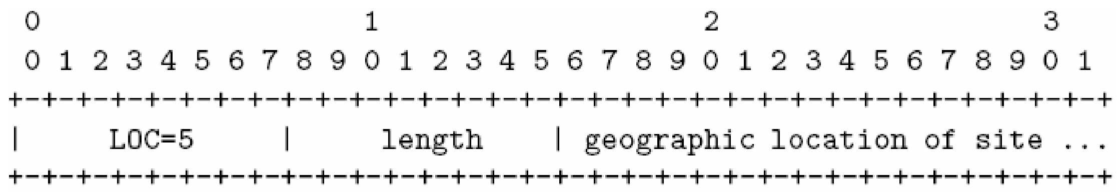


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    PHONE=4    |     length    | phone number of source      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12 – Format of PHONE item [2]

NAME item should be formatted as in Figure 12. The phone number of the source should be formatted with a plus sign in front replacing the international access code. For instance, it can be “+1 416 224 8736”, which is a phone number in Toronto, Canada.

■ **LOC:** Geographic User Location SDES Item

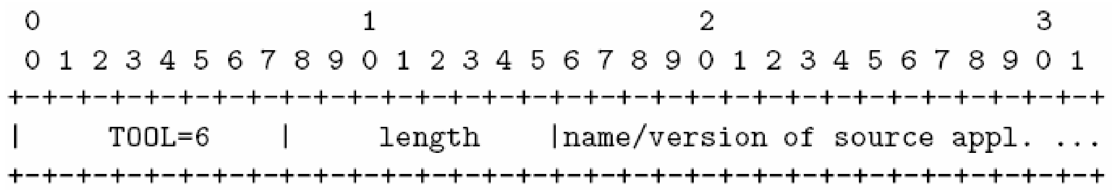


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     LOC=5     |     length    | geographic location of site ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 13 – Format of LOC item [2]

LOC item should be formatted as in Figure 13. The degrees of details depend on the applications and should be left to the application users. This item should remain unchanged for the lifetime of the session, except for mobile participants.

- **TOOL:** Application or Tool Name SDES Item

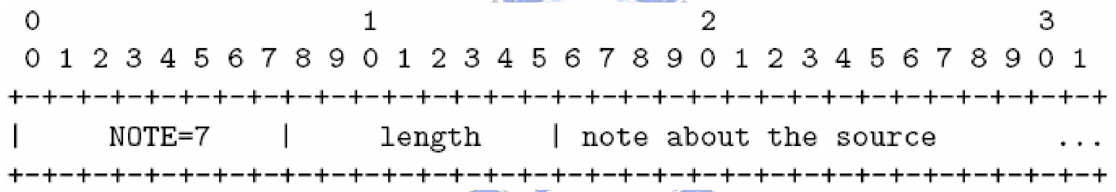


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    TOOL=6     |     length    |name/version of source appl. ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 14 – Format of TOOL item [2]

TOOL item should be formatted as in Figure 14. The name/version should be those of the application that is generating the stream, for example, “Skype”, and should remain unchanged for the lifetime of the session.

- **NOTE:** Notice/Status SDES Item

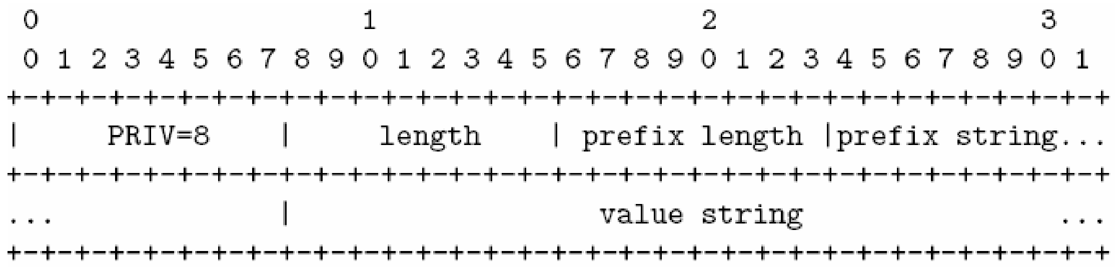


```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    NOTE=7     |     length    | note about the source       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 15 – Format of NOTE item [2]

NOTE item should be formatted as in Figure 15. It describes the current status of the source, for example, “busy” or “active”.

- **PRIV:** Private Extensions SDES Item



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    PRIV=8     |     length    | prefix length |prefix string...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
...             |                  value string               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 16 – Format of PRIV item [2]

PRIV item should be formatted as in Figure 16. It is used to define trial or application-specific SDES functionalities. It contains a prefix including a length-string pair. The prefix length field describes the length of the prefix string, which is a name chosen to be unique with respect to other PRIV items that might be received. The value string field carries the desired data.

BYE

```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |V=2|P|    SC   |   PT=BYE=203  |             length            |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                           SSRC/CSRC                           |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       :                              ...                              :
       +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 (opt) |     length    |               reason for leaving            ...
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 17 - Format of BYE [2]

A BYE RTCP packet signifies one or more sources' terminations. Referring to Figure 17, it contains the following fields:

- **Version (V), Padding (P), Length:**

  - They are the same as those defined in the format of SR packet.
- **Packet Type (PT):** 8 bits

  - Its value is set to 203 to identify this as a BYE packet.
- **Source Count (SC):** 5 bits

  - The number of SSRC/CSRC identifiers included in this BYE packet.

APP

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P| subtype |   PT=APP=204  |             length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           SSRC/CSRC                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          name (ASCII)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   application-dependent data                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 18 - Format of APP [1]

The APP packet type is meant for trial use as new features or new applications are developed. It should be formatted as in Figure 18. It contains the following fields:

- **Version (V), Padding (P), Length:**
  - They are the same as those defined in the format of SR packet.
- **Subtype:** 5 bits
  - It is used to permit a set of APP packets to belong to a new defined group.
- **Packet Type (PT):** 8 bits
  - Its value is set to 204 to identify this as an APP packet.
- **Name:** 4 octets
  - This field contains the name chosen by the creator of this application to be unique with respect to other APP packets that this application might receive.
- **Application-dependent Data:** variable length
  - These data may or may not appear in an APP packet and must be a multiple of 32 bits long.

2.2 The Real Time Streaming Protocol

To save the cost of storing multimedia data in large storage devices, such as hard drives, and to reduce the time to wait for multimedia files to be downloaded completely, streaming technology is used widely today. It allows users to enjoy multimedia content while downloading. Typically, the real-time multimedia data are compressed by content providers (servers), transmitted through the network, and decompressed and playbacked by the content subscribers (clients). The flow is just like a water stream and this is why this technology is called “streaming”. Clients can play the first received packet, decompress the second, while receiving the third and the forth; thus, clients can start enjoying the multimedia without waiting for the completion of data downloading.

The Real Time Streaming Protocol (RTSP) [3] is an application-level and client-server multimedia presentation control protocol, which enables controlled deliveries of single or multiple time-synchronized streams of real-time multimedia data such as video and audio, using the Transmission Control Protocol (TCP) or the User Data Protocol (UDP). It was developed to tackle the needs for efficient delivery of streamed multimedia over IP networks. It provides VCR-style remote control functionalities, like pause, fast forward, reverse, and absolute positioning, to control the streams of data from content providers. It provides methods for choosing delivery channels and delivery mechanisms based upon RTP. It supports both unicast for a single user and multicast for a large number of participants.

2.2.1 Supporting Operations

Quoted from [3], RTSP supports the following operations:

- Retrieval of media from content provider
  - If the client demands a media presentation, it can request a presentation description via HTTP.
- Invitation of a media server to a conference
  - If the contents of a media server are desired by the participants in an existing conference, the server can be invited to the conference by the participants.
- Addition of media to an existing presentation
  - The server can inform the client about additional available media becoming available. If the informed media are desired by the client, they can be added to the existing RTSP session.

2.2.2 Properties

Referring to [3], RTSP has the following elegant properties:

- Extendable
  - New functions can be added to RTSP effortlessly.
- Easy to parse
  - RTSP can be parsed by standard HTTP or MIME parsers.
- Secure
  - RTSP employs current web security mechanisms at the transport level and within the protocol itself. All existing HTTP authentication mechanisms are applicable.
- Transport-independent
  - Since RTSP runs on application-level, it does not rule out the use of an unreliable datagram protocol (UDP), or a reliable stream protocol such as TCP.
- Multi-server capable

- The client can automatically establish several simultaneous control sessions with different media servers. Media synchronization is performed at the transport level such as RTP.

- Separation of stream control and conference initiation
  - Stream control is detached from inviting a media server to a conference.
- Suitable for professional applications
  - RTSP supports frame-level accuracy and allows remote digital editing.
- Presentation description neutral
  - The protocol can convey the type of format to be used and does not require a particular presentation description which, however, must include at least one RTSP URI.
- Proxy and firewall friendly
  - The protocol should be readily handled by both application and transport layer firewalls.
- HTTP-friendly
  - As RTSP reuses HTTP concepts, the existing infrastructure can be utilized.
- Appropriate server control
  - If a client can begin a stream, it must be able to stop a stream. Servers should not start streaming to clients if clients cannot stop the stream.
- Transport negotiation
  - Before the client actually needs a media stream, it can negotiate the transport method.
- Capability negotiation
  - The client will be informed via an easy mechanism if any basic feature is missing.

2.2.3 RTSP States

According to [3], media data flow and RTSP controls do not have to be sent via the same protocol. For instance, RTSP controls may run on TCP while the data may be transmitted via RTP. In fact, if there is no more RTSP control is issued by the client to the server, the data flow will still continue until the end of the presentation. In addition, RTSP requests during the lifetime of a media stream may be from different TCP connections.

Therefore, it is essential for both the server and the client to maintain “session state” to be able to associate RTSP requests and responses, respectively, with a stream.

Referring to Figure 19, note that the rectangles represent methods which cause state changes if necessary, and the ellipses denote various states. The methods are described in the next section. Initially, the server (or client) is in “Init” state. After receiving “SETUP” request (response), it will transit to “Ready” state. Sequentially, it will receive “PLAY” request (response) so the streaming process can be started (“Playing” state). In addition, it the streaming process needs to be halted for some reason, it will receive “PAUSE” request (response) and go back to “Ready” state.

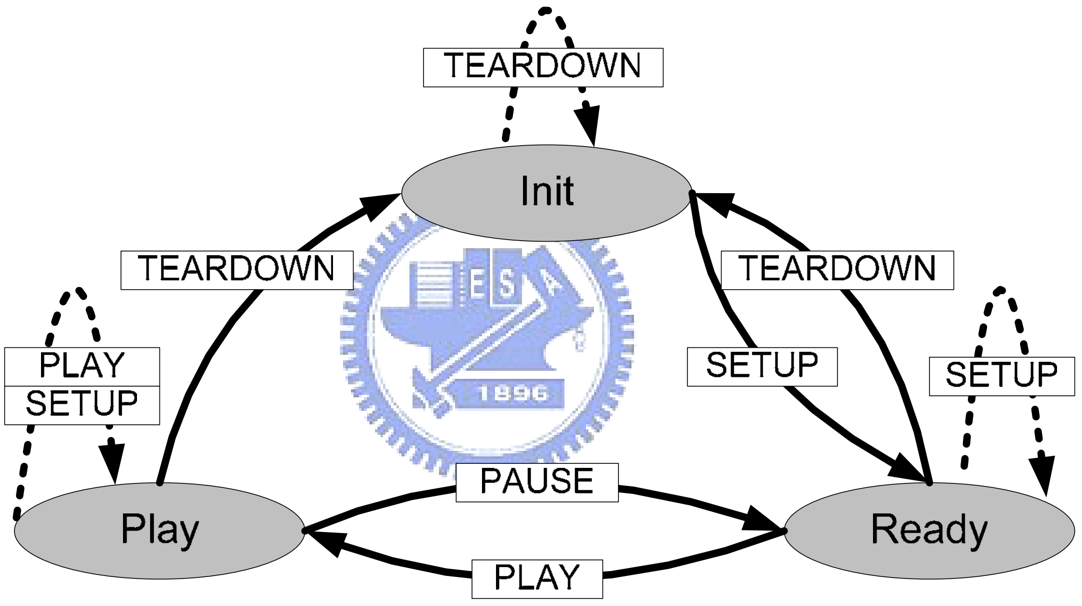
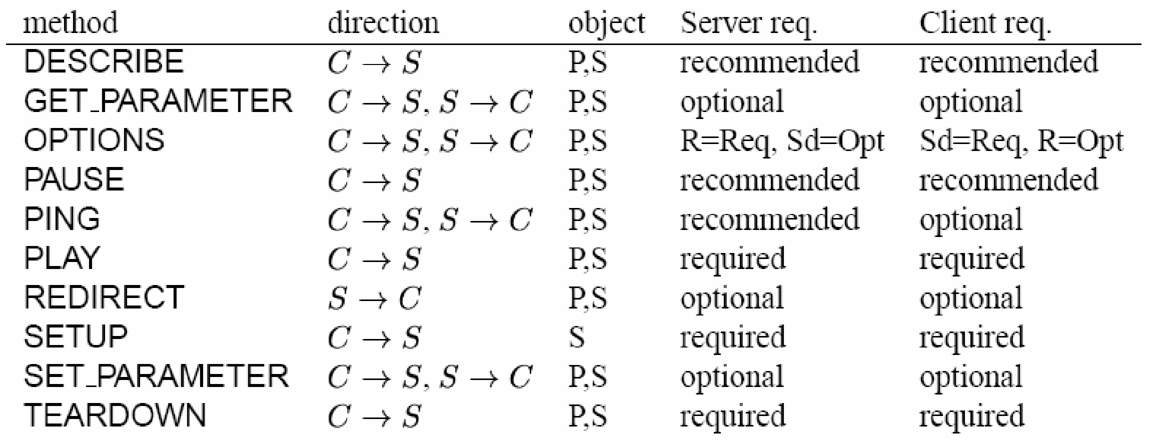


Figure 19 – RTSP State Machine

2.2.4 Method Definitions

A summary of available RTSP methods are listed in Table 1 [3]. Note that “direction” field indicates that in what direction the request should be sent. Also note that the field “object” specifies what objects they operate on. The field “Server req.” or “Client req.” states whether a method is required to be supported, optional, or recommended by the server or the client, respectively.

Table 1 – Overview of RTSP methods, their directions, and what objects they operate on. Legend: P=presentation, S=stream, R=Responds to, Sd=Send, Opt: Optional, Req: Required, Rec: Recommended [3].



method	direction	object	Server req.	Client req.
DESCRIBE	$C \to S$	P,S	recommended	recommended
GET_PARAMETER	$C \to S, S \to C$	P,S	optional	optional
OPTIONS	$C \to S, S \to C$	P,S	R=Req, Sd=Opt	Sd=Req, R=Opt
PAUSE	$C \to S$	P,S	recommended	recommended
PING	$C \to S, S \to C$	P,S	recommended	optional
PLAY	$C \to S$	P,S	required	required
REDIRECT	$S \to C$	P,S	optional	optional
SETUP	$C \to S$	S	required	required
SET_PARAMETER	$C \to S, S \to C$	P,S	optional	optional
TEARDOWN	$C \to S$	P,S	required	required

- **DESCRIBE**
  - It requests information about a presentation. For example [3], if the client wants the information about "rtsp://server.example.com/fizzle/foo" and it understands three types of description formats, "application/sdp", "application/rtsl", and "application/mheg", it can issue a request as:

    Request from client to server:

```
DESCRIBE rtsp://server.example.com/fizzle/foo RTSP/1.0
CSeq:       312
User-Agent: PhonyClient 1.2
Accept:     application/sdp, application/rtsl, application/mheg
```

    Response from server to client:

```
RTSP/1.0 200 OK
CSeq:       312
Date:       23 Jan 1997 15:35:06 GMT
Server:     PhonyServer 1.0
Content-Type: application/sdp
Content-Length: 376
v=0
o=mhandley 2890844526 2890842807 IN IP4 126.16.64.4
s=SDP Seminar
i=A Seminar on the session description protocol
```

```
u=http://www.cs.ucl.ac.uk/staff/M.Handley/sdp.03.ps
e=mjh@isi.edu (Mark Handley)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 3456 RTP/AVP 0
m=video 2232 RTP/AVP 31
m=application 32416 UDP WB
a=orient:portrait
```

The server then replies to the client with the information formatted in the type "application/sdp".

- **GET_PARAMETER**
  - It inquires about the value of a parameter of a presentation. For example [3], if the server wants to know the values of "packets_received" and "jitter", it can issue a request like the following:

    Request from server to client:

```
GET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
CSeq:      431
Content-Type: text/parameters
Session:   12345678
Content-Length: 15
packets_received
jitter
```

    Response from client to server:

```
RTSP/1.0 200 OK
CSeq: 431
Content-Length: 46
Content-Type: text/parameters
packets_received: 10
jitter: 0.3838
```

- **Options**
  - It queries what methods are supported. For example [3]:

    Request from client to server:

```
OPTIONS * RTSP/1.0
CSeq:          1
User-Agent:    PhonyClient/1.2
Require:
Proxy-Require: gzipped-messages
Supported:     play-basic
```

Response from server to client:

```
RTSP/1.0 200 OK
CSeq:          1
Public:        DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE
Supported:     play-basic, implicit-play, gzipped-messages
Server:        PhonyServer/1.0
```

- **PAUSE**
  - It halts the stream delivery of a presentation. For example [3]:

    Request from client to server:

    ```
    PAUSE rtsp://example.com/fizzle/foo RTSP/1.0
    CSeq:     834
    Session:  12345678
    ```

    Response from server to client:

    ```
    RTSP/1.0 200 OK
    CSeq:     834
    Date:     23 Jan 1997 15:35:06 GMT
    Range:    npt=45.76-
    ```

- **PING**
  - It may be used by both the server and the client to check each other's liveness. For example [3], if the client wants to perform a single-hop liveness check, and the server responds with a positive result, then they should have a conversation like the following:

    Request from client to server:

    ```
    PING * RTSP/1.0
    CSeq:     123
    Session:  12345678
    ```

Response from server to client:

```
RTSP/1.0 200 OK
CSeq:      123
Session:   12345678
```

- **PLAY**
  - It informs the server to start transmitting the requested data. For example [3], if the client only wants to enjoy the contents from 10 to 15 seconds, 20 to 25 seconds, and 30 seconds to the end, it can request the server as the following”

    Request from client to server:

```
PLAY rtsp://audio.example.com/twister.en RTSP/1.0
CSeq:      833
Session:   12345678
Range:     npt=10-15, npt=20-25, npt=30-
```

- **REDIRECT**
  - It informs the client that it has to connect to another server. For example [3], if the server wants to redirect the traffic to a new server at a specific time, it can issue a request as the following:

    Request from server to client:

```
REDIRECT rtsp://example.com/fizzle/foo RTSP/1.0
CSeq:      732
Location:  rtsp://bigserver.com:8001
Range:     npt=0- ;time=19960213T143205Z
Session:   uZ3ci0K+Ld-M
```

- **SETUP**
  - It specifies the transport mechanism for the streamed media. In the following example [3], the client wants to create an RTSP session containing the media, “rtsp://example.com/foo/bar/baz.rm” and accepts two types of transport mechanisms: RTP/AVP/UDP received on client port 4588 and 4589, and RTP/AVP/TCP received on RTSP control channel. The sever answers with the response indicating that it prefers the first acceptable transport mechanism, that it

will send and receive RTP and RTCP via ports 6256 and 6257 respectively, and that it will use “2A3F93ED” as its SSRC.

Request from client to server:

```
SETUP      rtsp://example.com/foo/bar/baz.rm RTSP/1.0
CSeq:      302
Transport: RTP/AVP/UDP;unicast;client_port=4588-4589,
           RTP/AVP/TCP;unicast;interleave=0-1
```

Response from server to client:

```
RTSP/1.0 200 OK
CSeq:      302
Date:      23 Jan 1997 15:35:06 GMT
Server:    PhonyServer 1.0
Session:   47112344
Transport: RTP/AVP;unicast;client_port=4588-4589;
           server_port=6256-6257;ssrc=2A3F93ED
Accept-Ranges: NPT
```

- **SET_PARAMETER**
  - Opposite to GET_PARAMETER, it sets the value of a parameter of a presentation. For example [3]:

    Request from client to server:

```
SET_PARAMETER rtsp://example.com/fizzle/foo RTSP/1.0
CSeq: 421
Content-length: 20
Content-type: text/parameters
barparam: barstuff
```

Response from server to client:

```
RTSP/1.0 451 Parameter Not Understood
CSeq: 421
Content-length: 10
Content-type: text/parameters
barparam
```

- **TEARDOWN**
  - It ends the session by stopping the stream delivery and freeing the resources

associated with the session. For example [3], if the client wants to end the session "rtsp://example.com/fizzle/foo" and the server grants the request, then they will have the following dialogue:

Request from client to server:

```
TEARDOWN rtsp://example.com/fizzle/foo RTSP/1.0
CSeq:      892
Session:   12345678
```

Response from server to client:

```
RTSP/1.0 200 OK
CSeq:      892
Server:    PhonyServer 1.0
```

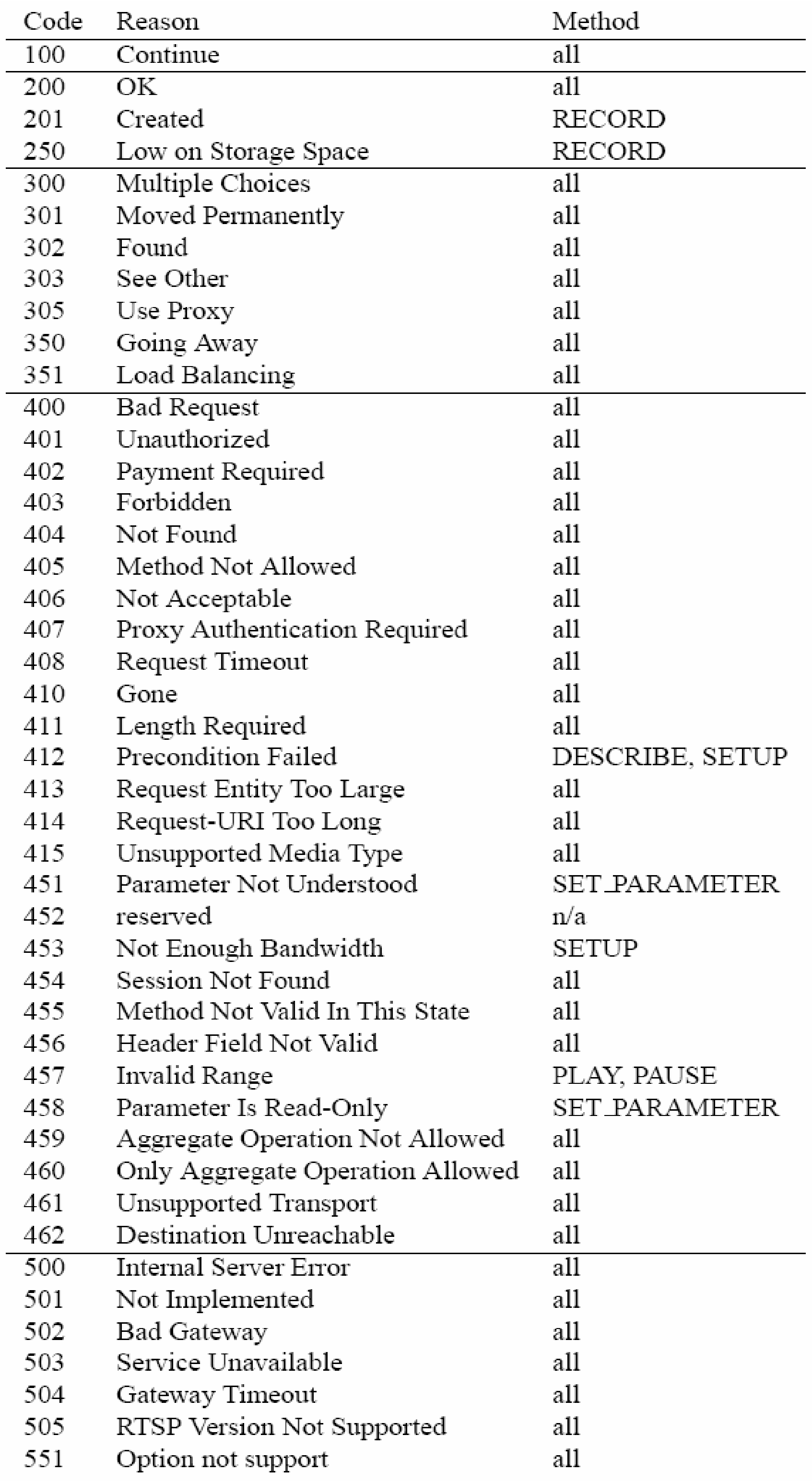
2.2.5 RTSP Status Codes

In an RTSP response message corresponding to an RTSP request message, a numerical value named "status code" is included to give the result of the attempt to understand and satisfy the request. The first digit of the status code defines the class of the response, which is one of the followings:

- 1xx: Informational
  - It indicates that the request was received and the process continues.
- 2xx: Success
  - It indicates that the request was successfully received, understood, and accepted.
- 3rr: Redirection
  - It indicates that a further action must be performed so the request can be completed.
- 4xx: Client Error
  - It indicates that the request contains syntax-error or cannot be accomplished.
- 5xx: Server Error
  - It indicates that the server was not able to accomplish a valid request

In Table 2, all available status codes and their usage with RTSP methods are listed.

Table 2 – Status codes and their usage with RTSP methods [3]



Code	Reason	Method
100	Continue	all
200	OK	all
201	Created	RECORD
250	Low on Storage Space	RECORD
300	Multiple Choices	all
301	Moved Permanently	all
302	Found	all
303	See Other	all
305	Use Proxy	all
350	Going Away	all
351	Load Balancing	all
400	Bad Request	all
401	Unauthorized	all
402	Payment Required	all
403	Forbidden	all
404	Not Found	all
405	Method Not Allowed	all
406	Not Acceptable	all
407	Proxy Authentication Required	all
408	Request Timeout	all
410	Gone	all
411	Length Required	all
412	Precondition Failed	DESCRIBE, SETUP
413	Request Entity Too Large	all
414	Request-URI Too Long	all
415	Unsupported Media Type	all
451	Parameter Not Understood	SET_PARAMETER
452	reserved	n/a
453	Not Enough Bandwidth	SETUP
454	Session Not Found	all
455	Method Not Valid In This State	all
456	Header Field Not Valid	all
457	Invalid Range	PLAY, PAUSE
458	Parameter Is Read-Only	SET_PARAMETER
459	Aggregate Operation Not Allowed	all
460	Only Aggregate Operation Allowed	all
461	Unsupported Transport	all
462	Destination Unreachable	all
500	Internal Server Error	all
501	Not Implemented	all
502	Bad Gateway	all
503	Service Unavailable	all
504	Gateway Timeout	all
505	RTSP Version Not Supported	all
551	Option not support	all

2.2.6 A Typical RTSP Unicast Session Example

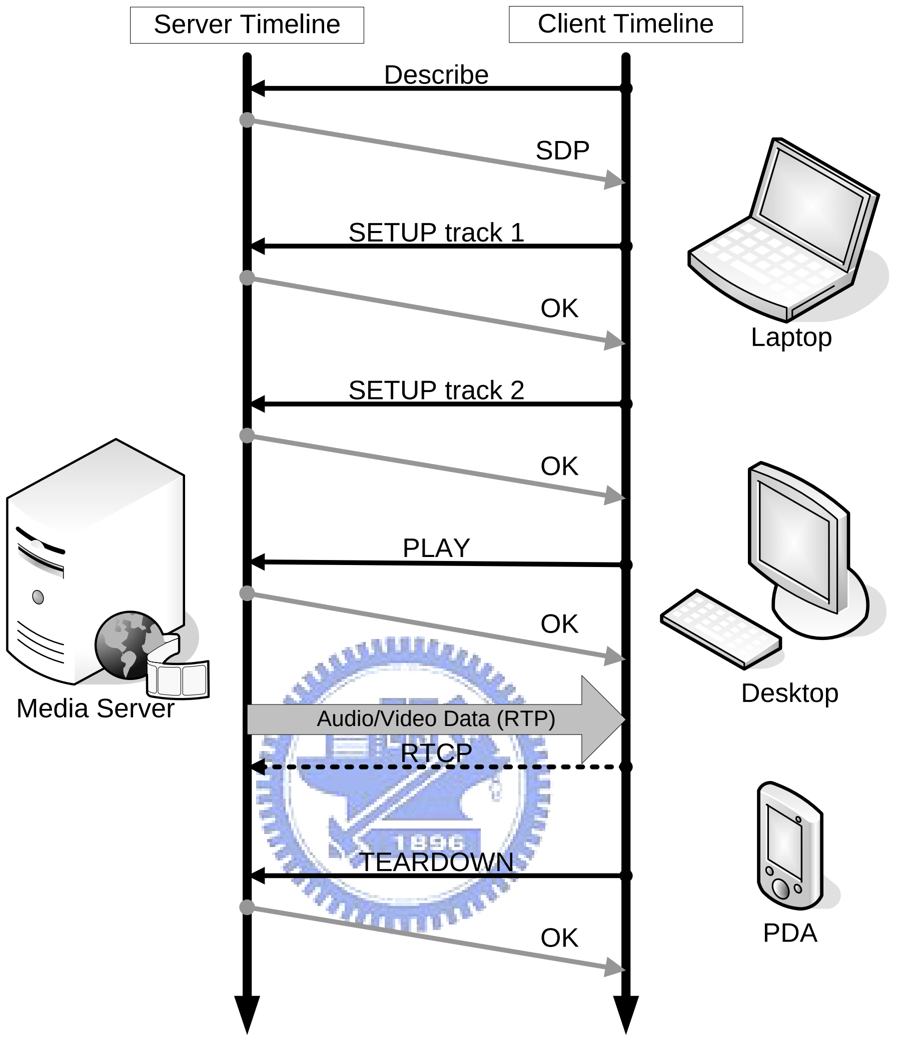


Figure 20 – A typical RTSP unicast session example

To summarize what have been described, an example is illustrated in Figure 20. As can be seen in the figure, a client (possibly a laptop, a desktop, or a PDA) wants to establish a connection with the media server in order to enjoy the media data. The first thing that the client has to do is to issue the request “DESCRIBE”, which asks the server for the information about the desired presentation. The server then replies with the information in accordance with Session Description Protocol (SDP) [4]. The client continues the action by issuing two “SETUP” requests to setup media data “track 1” and “track 2”. Since these two requests are valid and understood by the server, the server replies with “OK” to the client for both requests. After the setup procedure, the client wants to start to enjoy the media data, so it proceeds with issuing “PLAY” request. The server agrees with the request by replying “OK”, and starts to stream the data via RTP channel. While transmitting the data, the client may simultaneously send reports about its reception statistics to the server via RTCP channel. In this way, the server can adjust its sending rate or change its transmission method when the RTP channel bandwidth alters. If the client wants to end the presentation, the client can issue a “TEARDOWN” request indicating the end of the session so the server can stop streaming the data.

2.3 Introduction to Traffic Smoothing Algorithms

Nowadays, numerous multimedia streaming applications like video-on-demand services, transmissions of recorded sports events or concerts, and distant learning depend on the efficient transfer of prerecorded video. Content servers usually store video on large and fast disks [5]－[7]. Clients such as PCs or set-top boxes, which have a limited-size playback buffer for storing temporary media data, can connect to content servers through IP networks.

If high-quality videos are demanded, they will still require a huge amount of storage space and transmission bandwidth, even when they are already compressed by the most advanced encoding technologies such as MPEG-4 or H.264. These modern encoding schemes generate bitstreams of constant quality but variable-bit-rate (VBR) to avoid the quality degradation caused by constant-bit-rate (CBR) encoding methods [8] [9]. However, this VBR characteristic can cause significant burstiness on multiple time scales due to the natural variations within and between scenes, and the frame structure of the encoding algorithm [10], and as a result, complicates the allocation of disk, memory, and network resources [11]. Therefore, to be able to efficiently transmit constant-quality video, a technique that can smooth the burstiness is required. Intuitively, one may suggest that the server, network, and client could allocate resources based on the peak bitrate of the video, such over-reserving is exceedingly wasteful and depreciates the benefits of a constant-quality encoding.

Fortunately, prerecorded video gives us an opportunity to reduce the variability of the network bandwidth requirements by transmitting frames to the client playback buffer in advance of each burst [12]. Based on a priori knowledge of the size of each frame, the server can precompute a transmission plan that minimizes the bitrate variation and peak bitrate while avoiding both underflow and overflow of the client's buffer. This technique is what we call "bandwidth smoothing". Its benefits are the results of removing both short-term burstiness (e.g. large size difference between I-frames and P-frames) and medium term burstiness within and between scenes [13].

So far, in this field, six major smoothing algorithms have been proposed [13]－[18]. They have a common primary goal that is to reduce the peak rate of the stream, but however, they differ in what performance metrics they attempt to optimize. Hence, they can generate different transmission schedules of different performance properties, such as minimum number of rate changes in transmission, minimum variability of the bandwidth requirements, minimum utilization of the client buffer, minimum number of on–off segments in an on–off transmission model, change of transmission rates only at periodic intervals, and minimum general cost metrics through dynamic programming.

For the implementation of the system concerned in this paper, since MVBA [13] achieves the goals of minimizing the peak transmission rate and the rate variability, and it is easier to implement than other algorithm, it will be selected among six smoothing algorisms to help the server transmit more efficiently.

2.3.1 MVBA

Goals

The goals of MVBA, as claimed by the authors [13], are that given a fixed-size client buffer, a feasible transmission plan with:

- minimum variance of transmission rate
- minimum peak transmission rate
- and maximum lowest transmission rate

will be computed based on the *a prior* knowledge of frame sizes in the stored video. Note that a feasible transmission plan is a transmission plan that ensures the client buffer will never be drained nor overflowed. If the client buffer starves, then the presentation will be held up until more data are received. Conversely, if the client's buffer is overloaded with data, then all the lately arriving data will be discarded until some spaces of the buffer become available.

Main Concepts

- Intuitively, because CBR transmission is the smoothest possible, during the computation of the transmission plan, CBR transmission segments should be extended as far as possible.
- When the transmission rate must be increased or decreased to avoid starving or overflowing the client buffer, respectively, this rate change should be taken place as early as possible as to ensure the minimum rate variability.

Therefore, with MVBA, the transmission rate changes always occur at the earliest possible times for both rate increases and decreases. Thus, the generated transmission plan indeed gradually modifies the bandwidth requirement of the stream.

Concept Realization

Up to now, only the goals of MVBA and the ideas behind MVBA have been introduced. How MVBA can be implemented in a real server system is still unmentioned and should be described now. In [13], the authors provide a complete pseudo code and ways that can generate an MVBA transmission schedule. The following notations are worth knowing [13]:

- C_{max}: the maximum transmission rate at which the server can stream over a given interval, without overflowing the client buffer.
- C_{min}: the minimum transmission rate at which the server can stream over a given interval, without draining the client buffer.

In fact, the resulted transmission schedule will consist of a series of CBR transmission segments that might be C_{max} or C_{max}. Thus, to implement MVBA in a real server system, the system developer will just have to re-write the pseudo code in the used programming language such as C or C++. The resulted transmission plan will be the number of bytes that should be sent at each time instants. For example, for a video with the framerate of 25 frames per second, then the transmission plan probably will specify that at 0 ms, 1666.67 bytes should be transmitted, at 40 ms, 566.67 bytes should be transmitted, at 80 ms, another 333.33 bytes should be transmitted, and so on.

2.3.2 Packet-based MVBA

It is well known that in the current model of Internet, the data being transmitted from the source to the destination are not injected into the network bit by bit or byte by byte. Instead, in the real world, the data are sent as packets. Because along the path from the source to the destination, packets might pass by many different types of networks, each of which can support different maximum packet size (MTU), packets should not be made to be larger than the smallest MTU in these networks to avoid unnecessary fragmentations and reassembly. In addition to the maximum packet size, the size of each packet should be integers. A problem arises at this point. Recalling from previous section that at each time instant, the generated transmission plan will specify at each time instant how many bytes should be transmitted. The numbers of bytes specified may be of multiple decimal digits of precision or larger than the smallest MTU. To cope with this problem, a packet-based MVBA is needed and developed [24].

In short, the packet-based MVBA will generate a feasible transmission plan that follows the original MVBA generated plan the closest possible. The first step is to divide each video frame into fixed-size (say 1024 bytes) packets. For example, a frame of 3000 bytes will be segmented into two full 1024-byte packets and one 952-byte packet. Now instead of a series of frames available, a series of packets are on hand. Secondly, instead of just knowing the numbers of bytes that should be sent for different time instants as specified by the original MVBA, the accumulated number of bytes should also be computed. In other words, the total number of bytes that should be sent before and at each time instant is calculated. This new information will be referenced to derive the new packet-based plan.

Now, it is time to derive the new transmission schedule. Essentially, for the time instant belonging to the C_{max} segment in the original MVBA plan, the packet-based MVBA will take a number of packets from the series of available packets in order whose sizes sum up to be smaller or equal to the original accumulated number of bytes. This step ensures that the new schedule will approximate the old one the best and the client buffer will not be overflowed. Conversely, for the time instant belonging to the C_{min} segment in the original MVBA plan, the packet-based MVBA will take a number of packets from the series of available packets in order whose sizes sum up to be equal to or larger the original accumulated number of bytes. This step is to guarantee that the new schedule will follow the original one the closest possible and the client buffer will never starve.

This short section only describes the packet-based MVBA in brief. For a more complete and in depth discussions about it, readers are encouraged to refer to [24].

3 Some Programming Techniques

Traditional C/C++ programming techniques are not sufficient to provide enough functionality for system designers to implement an extremely complicated, sophisticated, and multi-functional system, such as the proposed client/server streaming system. Hence, more advanced programming techniques are needed. In this section, the notions of threads and synchronization objects are explained and discussed.

3.1 Threads

In a typical client/server system, both the server and the client programs have to be able to deal with many things concurrently. For example, the server program may have to transmit media data to several clients at the same time, receive RTSP requests from those clients simultaneously, and manages interactions with the user (keyboard and mouse input). However, since a typical C/C++ program usually executes the instructions from top to bottom, possibly with several function calls, there is no notion of multitasking. This kind of program is called "single process, single thread". To be able to handle multiple tasks concurrently, the programming technique," single process, multiple threads" or multitasking, should be employed.

To define more formally, each process is provided by the operating system with the resources needed to execute a program. A process owns a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a base priority, minimum and maximum working set sizes, and at least one thread of execution. In addition, each process is started with a single thread (primary thread); however, it can create additional threads from any of its threads when needed. Threads are the entities within a process that can be scheduled for execution, and they share the virtual address space of the process and system resources. Moreover, each thread maintains its own exception handlers, scheduling priority, thread local storage, unique thread identifier, and set of structures the system will use to save the thread context, which includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the process[19]. Since Microsoft® Windows® supports preemptive multitasking, which creates the effect of simultaneous execution of multiple threads in a process, in this implementation of the client/server system, we will take advantages of it.

3.2 Mutex and Semaphores

Naturally, in a process that has multiple threads executing at the same time to achieve the team effort, these threads may want to communicate with each other in some way as to synchronize with or send information to each other. Intuitively, a global variable or array (shared memory) can be a good location to place shared messages or data since all threads can see and have access to it. However, a conflict occurs when two or more threads are writing new information to the shared memory. To prevent this problem, a synchronization object called mutex is needed to manage the shared memory. Each thread must wait for the ownership of a mutex before executing the code that accesses the shared memory, and after writing to the shared memory, the thread releases the mutex object. Normally, if more than one thread is waiting on a mutex, the thread that has been waiting in the queue (“first in, first out” or FIFO order) for the longest time is selected to own the mutex when it becomes available.

In the case when it is desirable that more than one and fewer than a maximum number of threads are allowed to access to the shared memory, a synchronization object called semaphore is needed. A semaphore maintains a count between zero and a specified maximum value. The count is decremented every time when a thread completes a wait for the semaphore and enters the critical section (the code segment where the thread executes the code that accesses the shared memory), and incremented each time when a thread releases the semaphore. If the count is zero, the thread that wishes to enter the critical section must wait until someone releases the semaphore. Let’s say if the count is three, then simultaneously three threads can access to the shared memory. When they do enter the critical section, the count will be reduced to zero. At this time, no more thread can access to the share memory. Thus, the semaphore acts as a gatekeeper that limits the number of threads sharing the resource to a specified maximum number. In this implementation, to control the rate of transmission, the scheduling thread can release the semaphore according to the schedule. When the transmission thread is executing too fast, it can be slowed down by the semaphore mechanism since it must wait for the count to become greater than one. If the transmission rate is too low, the transmission thread can be speeded up by incrementing the count of the semaphore.

3.3 Token Bucket

Traffic shaping allows one to control the traffic entering the network, and the rate at which the traffic is being sent. To implement this mechanism in a system, two major approaches are available: leaky-bucket and token-bucket schemes [20]. The former re-shapes bursty traffic to a steady one. The later realizes traffic shaping by dictating when and how much traffic can be inserted into the network based on the presence of tokens in the bucket. In the proposed system, the token bucket method is preferable and utilized to implement the smoothing algorithm, because the traffic that the streaming server generates will not be steady but variable according to a predefined schedule.

To illustrate the token bucket method more clearly, Figure 21 is provided. As depicted, the bucket contains tokens generated by the token generator, which produces tokens according to a predefined schedule (i.e. transmission plan). A token can represent that a flow is allowed to transmit a packet or a unit of bytes defined by the system designer. If the bucket is empty, a flow cannot transmit any data. Thus, the traffic shaping can be realized by defining a desired transmission plan that is to be used by the token generator to create tokens.

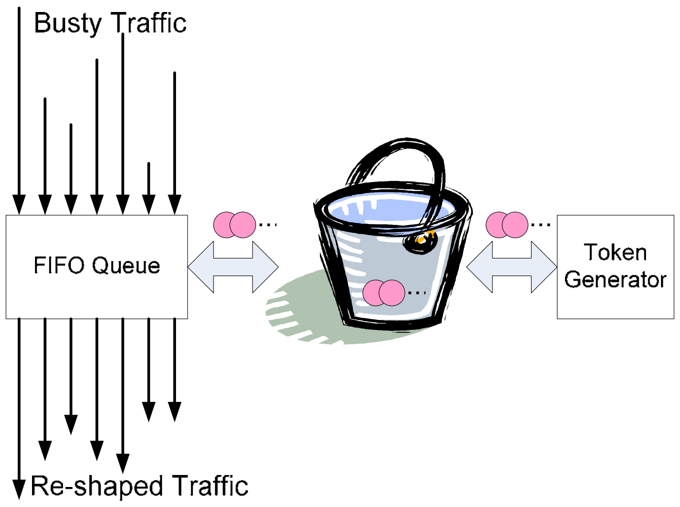


Figure 21 – Illustration of token bucket

4 Streaming System Implementation

In this chapter, how the proposed interactive RTP/RTSP streaming system is designed, developed, organized, and built will be explained in detail. Specifically, the system development environment, the expected system capabilities, the system architecture, and implementations of RTP streaming and RTSP controls will be explained in depth.

4.1 System Development Environment

The equipments on the following lists are needed when designing, developing, implementing, and executing the system. In addition, the hardware and software setup procedures are also provided.

4.1.1 Hardware Requirements

- One standard IBM-compatible PC for the server
- One standard IBM-compatible PC for the client
- One D-Link DSB-C320 USB 1.1 Web Camera
  - Supports up to 640x480 resolution
- Several SOHO Wireless Internet Camera
  - Supports IEEE 802.11b
  - Compresses images into Motion-JPEG format
  - Supports up to 640x480 resolution
- One D-Link DI-784 11a/11g Dualband 108 Mbps Wireless Router
  - Supports IEEE 802.11a, 802.11b, and 802.11g
  - Has built-in 4-port switch
  - Supports DHCP

4.1.2 Software Requirements

- Microsoft® Windows® 2000/NT/XP Operating System
- Microsoft® Visual C++ 6.0®
- MPEG-4 encoder/decoder libraries
- Software development kit for SOHO Wireless Internet Camera

- Motion-JPEG decoder library
- Viscom Software VideoCap Live v1.5 ActiveX Control
- D-Link DSB-C320 USB 1.1 Web Camera Driver

4.1.3 Hardware Setup

As depicted by Figure 22, several wireless cameras are connected to the streaming server via the wireless router either by the wireless or wired connection. In addition, a USB web camera is attached to the streaming server via USB 1.1 connection. The streaming clients can connect to the server to acquire either stored or live-captured MPEG-4 media files through the IP network.

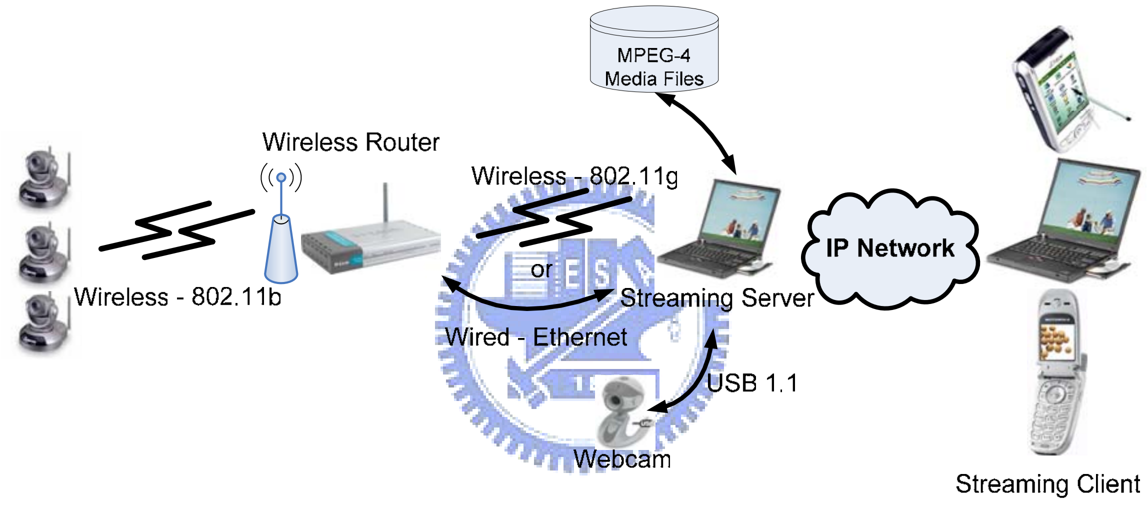


Figure 22 – Illustration of system hardware setup

4.1.4 Software Setup

Server PC

1. Install Microsoft® Windows® 2000/NT/XP.
2. Install Microsoft® Visual C++ 6.0®.
3. Install drivers for SOHO Wireless Internet Camera and D-Link DSB-C320 USB 1.1 Web Camera.
4. Install Viscom Software VideoCap Live v1.5 ActiveX Control.
5. When writing the program, remember to include the software development kit for SOHO Wireless Internet Camera, Motion-JPEG decoder library, and MPEG-4 encoder library.

Client PC

1. Install Microsoft® Windows® 2000/NT/XP.
2. Install Microsoft® Visual C++ 6.0®.
3. When writing the program, remember to include MPEG-4 decoder library.

4.2 Expected System Capabilities

Different implementations of streaming systems have diverse goals to achieve and expectations to fulfill. This applies to the proposed system implementation as well. Unlike the commercial streaming solutions, the proposed system does not focus on the performance of the codec used in the system and the friendliness of the graphical user interface (GUI) to the consumers. In fact, it concentrates on providing a more efficient transmission and simpler integration with image processing functions. The following sections will discuss, in detail, about what abilities that the server and the client are expected to possess.

4.2.1 Expected Capabilities of Server System

As depicted by Figure 23, in this work, the server system is designed to be capable of handling the following tasks:

1. The server should be able to accept and react to the user input. Particularly, when the user presses a button or checks a box, it should call the function specifically written to deal with this event.
2. The server should be able to accept the connection requests from clients.
3. The server, if requested by the program user, should be able to smooth its VBR traffic by using packet-based MVBA.
4. The server should be able to receive, recognize, parse, and verify RTSP requests sent by its clients. In addition, it should reply to its clients with corresponding RTSP responses when required.
5. The server should be able to perform RTSP state changes if necessary when receiving proper and valid RTSP requests sent by its clients.
6. The server should be able to packetize media data frames into fixed-size RTP packets.

7. The server should be able to parse MPEG-4 files. In particular, it must be able to open and close MPEG-4 files stored in its storage device, get frame data from them, read timestamps of frames, and so on.
8. When capturing audiovisual contents from any type of camera, the server should be able to compress them into MPEG-4 format.
9. After encoding the live-captured media contents into MPEG-4 format, the server must know how to save them into its storage device. In this way, when connecting to it, clients can see these MPEG-4 files available to them.
10. As described in Section 4.1, the selected wireless cameras compress video into MJPEG format. Thus, when the server receives MJPEG compressed data from these cameras, in order to transcode the video from MJPEG format to MPEG-4 format, it should be able to decode the MJPEG data to reconstruct the original signal.
11. The server should be able to acquire audiovisual contents from USB cameras connected to it.
12. The server should be able to acquire audiovisual contents from wireless cameras.
13. The server should be able to transmit RTP packets via UDP.
14. The server should be able to transmit RTP packets via TCP.

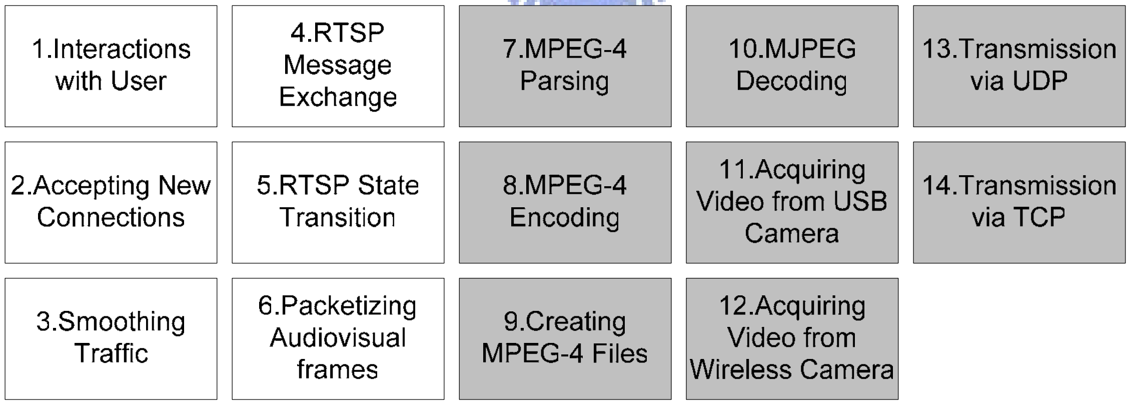


Figure 23 – Tasks that the server must handle in the proposed system

4.2.2 Expected Capabilities of Client System

As depicted by Figure 24, in this work, the client system is aimed to be capable of handling the following tasks:

1. The client should be able to accept and react to the user input. Particularly, when the

user presses a button or checks a box, it should call the function specifically written to deal with this event.

2. The client should be able to connect with the server.
3. The client should be able to de-packetize received RTP packets to reconstruct the original media data frames.
4. The client should be able to issue a sequence of RTSP requests to the server in order to establish a streaming session with the server and to receive desired data. In addition, it should be able to receive, recognize, parse, and verify RTSP responses replied by the server.
5. The client should be able to perform RTSP state changes if necessary when receiving proper RTSP responses replied by the server.
6. The client should be able to decode received MPEG-4 audiovisual data.
7. The server should be able to receive RTP packets from UDP.
8. The server should be able to receive RTP packets from TCP.

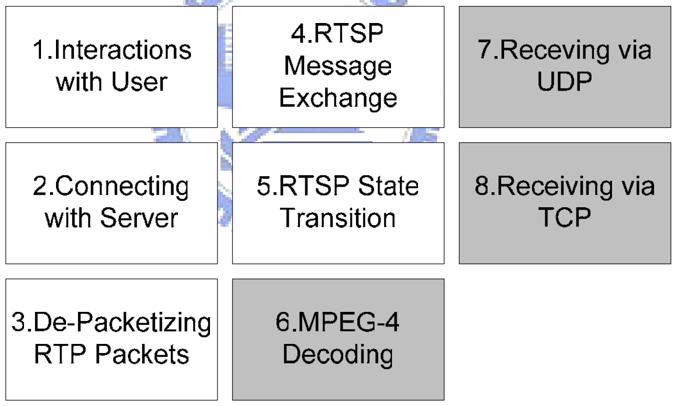


Figure 24 - Tasks that the client must handle in the proposed system

4.3 System Architecture

These server and client implementations allow one to setup multiple streaming sessions. The server can stream multimedia data to multiple clients in a real-time fashion while the client can receive and play the media contents transmitted by the server.

Figure 25 represents a typical client/system block diagram illustrating the uses of RTP and RTSP. A shown by the block on the bottom-left of the figure, the server can acquire images from wireless cameras or USB cameras, encode or transcode these acquired image data into MPEG-4 format. Then, the server will place these MPEG-4 bitstreams on the buffer, where these bitstreams will be waiting to be packetized by the packetizer into RTP packets and then placed onto the output packet buffer. Each time when an RTP packet becomes current (ready to be transmitted by the server and playbacked by the client), it will be transmitted to the client by the streamer, which if enabled by the RTCP mechanism, can adapt its transmission behavior to the network conditions. In addition, RTSP is often implemented on the top level, and is responsible for the streaming controls, e.g. SETUP, PLAY, PAUSE, STOP, and so on. The session controller maintains the RTSP state machine, deals with RTSP requests and responses, and reacts to the state transitions by performing necessary actions. On the left side of the figure is the client system. Basically it works similarly to the server but in a reverse direction. Upon receiving RTP packets, the receiver will place them onto the packet buffer, where the de-packetizer will collect and integrate them to reconstruct the original MPEG-4 bitstreams. The audio/video decoders will decode the bitstreams, so the original media contents can be rendered on any kind of display device and the speaker in a synchronized fashion. In our system, we follow closely to Figure 25 but do omit the RTCP mechanism, since the focus of this work is about streaming.

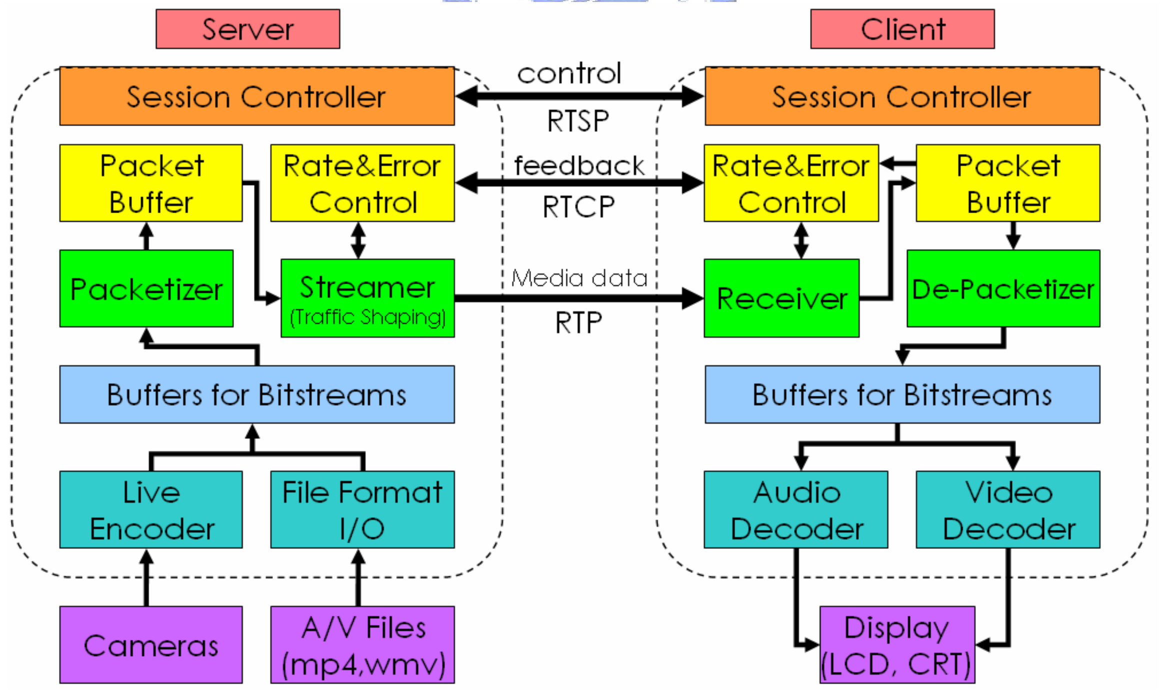


Figure 25 – Illustration of client/server system with RTP and RTSP in use

In the proposed client/server architecture, ten different modules are provided. They can be helpful and useful in creating multimedia streaming applications. The system designers who are developing the streaming server will need 3GP Parser module, 3GP Creator module, Sub Server module, Server Dialog module, and SP_encore library. The streaming client developers will need 3GP Parser module, Sub Client module, Server Dialog module, and SP_decore library. Since the MPEG-4 compression algorithms and their implementations are not the main subjects here, details about SP_decore and SP_encore libraries will not be described.

4.3.1Architecture of Streaming Server System

In the server workspace, there are five modules, each of which is responsible for some tasks shown in Figure 23 and is listed below.

1. 3GP Parser module
   - It is responsible for reading and parsing MPEG-4 compressed video files.
   - It handles the seventh task presented in Figure 23.
2. 3GP Creator module
   - It is responsible for creating MPEG-4 files (.mp4) and virtual files (.3gp) for clients to access live-captured audiovisual contents.
   - It handles the ninth task shown in Figure 23.
3. Sub Server module
   - It deals with all the matters about the transmission of streaming data on the server side, including the connection initiations, the connection negotiation, the connection establishments, the streaming of RTP packets, and RTSP related jobs.
   - It handles the second to sixth, and thirteenth to fourteenth tasks presented in Figure 23.
4. Server Dialog module
   - It is responsible for defining and creating the user interface for the server, and interacting with users.
   - It handles the first, and tenth to twelfth tasks shown in Figure 23.
5. SP_encore.lib library

- It is responsible for compressing the captured frames into the MPEG-4 format.
- It handles the eighth task in Figure 23.

How the five modules link to each other is depicted in Figure 26. The top block, the graphical user interface (GUI) represents the server dialogue. The MPEG-4 block is equivalent to the SP_encore.lib library. The gray blocks, as a whole, symbolize the sub server module. By comparing Figure 26 with Figure 25, it is not hard to realize that the architecture of the proposed server system mimics the typical one. However, they do differ in some parts. Beneficially, the streamer in the proposed server is enhanced by the bandwidth smoother, whose implementation details are given in the later section.

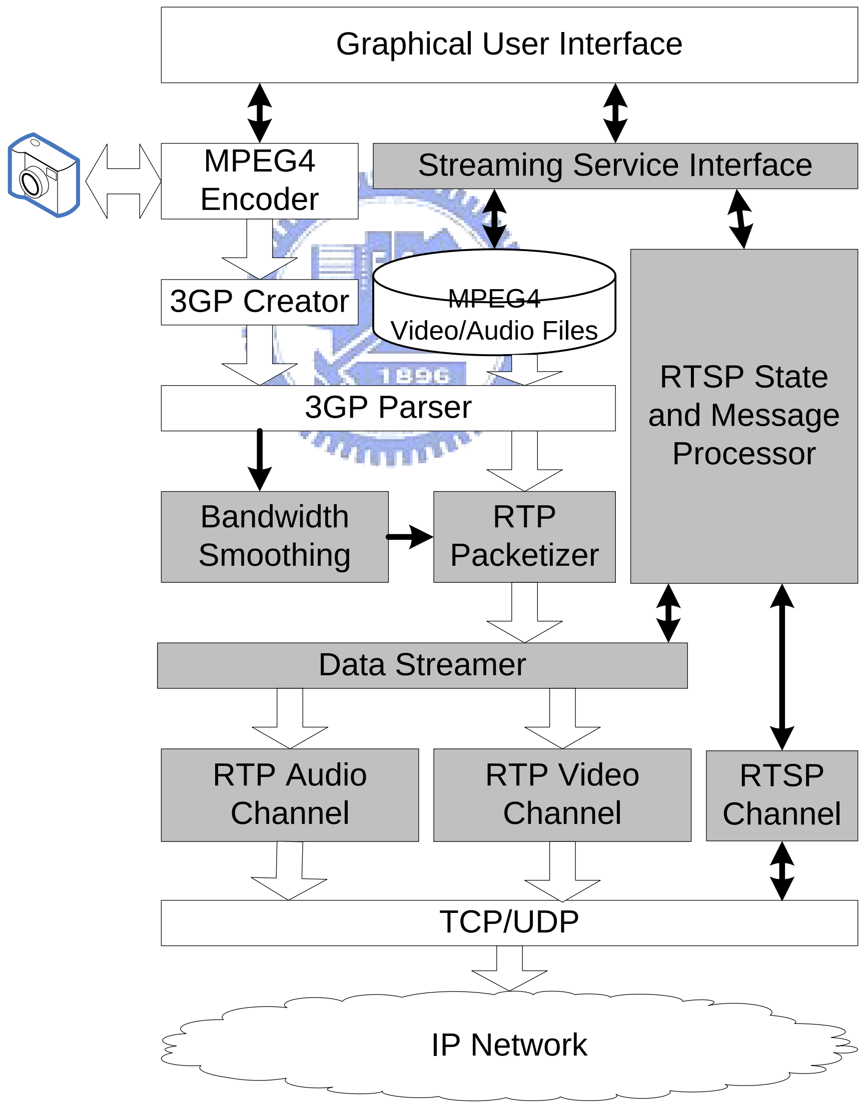


Figure 26 – Architecture of server system

Stepping a little bit deeper, on the top of Figure 26, the graphical user interface (GUI) is where the program will interact with the user. The user can trigger events by typing in texts in the boxes or pressing down the buttons provided in the GUI. For example, initially, the user can start the streaming service by pressing the start button. This event triggering can take place because the arrow pointing to both the GUI and the streaming service blocks provides means of communication between the two. Because at any time, a client may want to connect to the server in order to view the images captured by one of the cameras (wireless and web cameras) in real-time or stored in the server storage, the GUI block has to encode raw images acquired from the cameras into MPEG-4 format by using the MPEG-4 encoder. While compressing these images periodically and place them on a buffer, the server has to employ 3GP creator module in order to create a virtual file in the disk, so when a client connect to the server, the client will see that there are MPEG-4 files which are associated with different cameras available in the server storage. Whether the client is interested in the stored or live-captured audiovisual contents, the server will use the 3GP parser module to parse the MPEG-4 file (i.e. whether real or virtual, respectively) that the client demands. In turn, the 3GP parser will feed information to both the bandwidth smoothing and the RTP packetizer modules. To the former module, it will provide the presentation timing and the size of each compressed frames. With this information, the former module can calculate a smoother transmission plan according to the algorithm described in the earlier section. To the later module, the 3GP parser will not only provide the information given to the former one, but also offer a single compressed frame data to the packetizer whenever the packetizer is requesting one. The packetizer, as implied by its name, packetizes frames into fixed-size RTP packets in accordance with RTP [2]. In addition, strictly regulated by the bandwidth smoother, which follows the predefine transmission plan precisely, the packetizer feeds the RTP packets to the data streamer in a controlled manner. The data streamer will stream each RTP packet to the client via the established audio/video channels. If the user of the server presses the pause button, or the RTSP state and message processor receives a valid PAUSE request sent by the client via the RTSP channel, the processor will inform the data streamer to stop the streaming process. The data streamer will halt its transmission and perform necessary calculations in order to stream timely as the presentation is resumed.

Moreover, if the user presses the stop button, or the processor receives a valid STOP request, the processor will inform the data streamer to stop the streaming process and end the streaming session. The data streamer will terminate its transmission and release all the resources associated with the session.

4.3.1.1 Threads Used in Streaming Server System

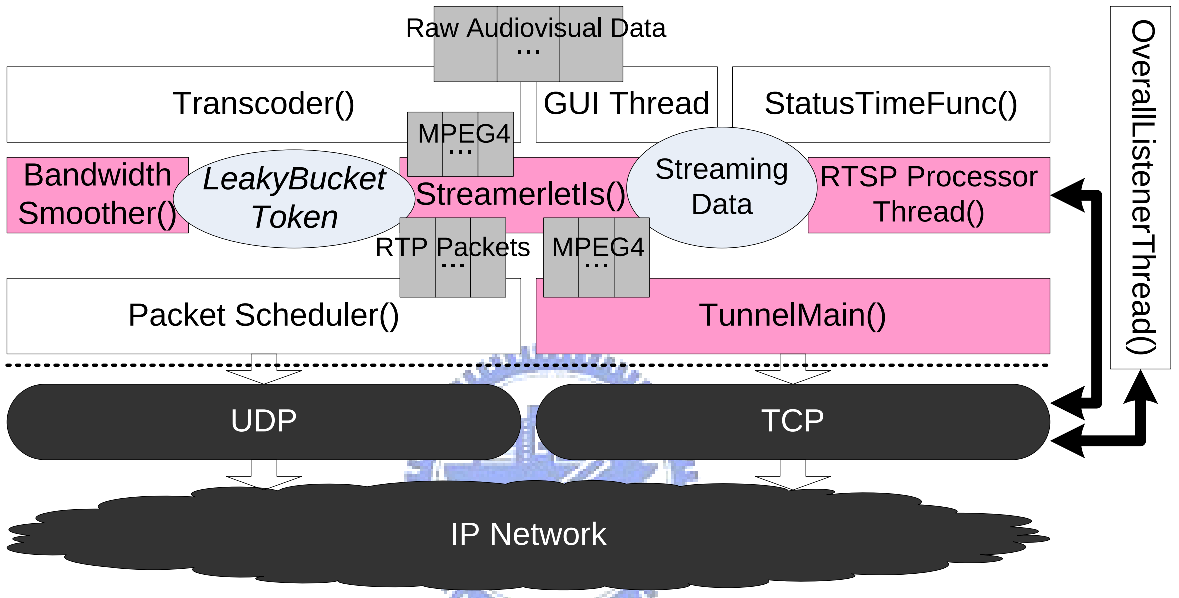


Figure 27 – Threads of server system

Having talked about the tasks and the architecture of the server system, it is time to explain how the system can be implemented in software. Intuitively, since lots of tasks have to be handled at the same time by the server program, multiple threads are definitely needed.

Indeed, referring to Figure 27, there are several threads running simultaneously in the server system when the connections are established with clients. Note that the overlaps between the gray ellipses/rectangles and threads (white rectangles) represent the relationships between the shared data (gray shapes) and the threads. Specifically, the following threads will execute once the server starts:

- GUI thread
  - Responsible for interacting with the user of the server program.
- *Transcoder()*

  - Responsible for acquiring audiovisual contents from various types of cameras, encoding or transcoding them into MPEG-4 format, and placing the resulted bitstreams onto the buffer where *StreamerletIs()* thread can access.
- *StatusTimeFunc()*
  - Responsible for collecting and displaying all the clients' statuses on the GUI.
- *OverallListenerThread()*
  - Responsible for accepting new incoming connections. Every time a new connection is accepted and established successfully, it will create an instance of *"RTSPProcessorThread()"*, which from this point, will take control of this connection.
- *PacketScheduler()*
  - Responsible for injecting RTP packets into network via UDP sockets. It is the data streamer for transmission over UDP.

However, the following group of threads will start only when a client is connecting to the server. For each new RTSP session, the system will start a new group of these threads.

- *RTSPProcessorThread()*
  - Responsible for accepting, parsing, extracting, and replying the RTSP messages, and performing necessary state transitions. It is equivalent to the session controller shown in Figure 25.
- *StreamerletIs()*
  - Responsible for packetizing each data frame into RTP packets if UDP is used as the transmission protocol. It acquires frames from the target MPEG-4 file using 3GP parser module and packetizes them according to the predefined schedule for the requested audiovisual content. It places packetized RTP packets onto the buffer where "*PacketScheduler()*" can access. On the other hand, it TCP is selected as the transmission protocol, this thread will also acquire frames from the target MPEG-4 file according to the predefined schedule for the requested audiovisual content, but however, will not packetize them. Then it will place these acquired frames onto the buffer where "*TunnelMain*()" can access.
- *BandwidthSmoother()*

- Responsible for controlling the frequency of packetizing a new frame performed by StearmerletIs(), so the transmission behavior (rate) of the data streamer can be adjusted to improve the overall transmission performance.

Finally, the following thread will be created only when a client who demands that the RTP packets are to be transmitted over TCP is connecting to the server. For each new RTSP session, the system will start a new one.

- *TunnelMain()*
  - Responsible for injecting media data frames into network via TCP sockets. It is the data streamer for transmission over TCP.

4.3.2 Architecture of Streaming Client System

In the client workspace, there are three modules, each of which is responsible for some tasks shown in Figure 24 and is listed below.

1. Sub Client module
   - It deals with all the matters about the transmission of streaming data on the client side, including the connection initiations, the connection negotiation, the connection establishments, the receiving of RTP stream, and RTSP related jobs.
   - It handles the second to fifth, and seventh to eighth tasks presented in Figure 24.
2. Client Dialog module
   - It is responsible for defining and creating the user interface for the client, and interacting with users.
   - It handles the first task in Figure 24.
3. SP_decore.lib library
   - It is responsible for decompressing the MPEG-4 compressed frames.
   - It handles the sixth task in Figure 24.

The architecture of the client system is much simpler than that of the server system. Referring to Figure 28, the top block, the graphical user interface (GUI) represents the client dialogue module. Following that is a group of gray blocks, which as whole, are the sub client module. The MPEG-4 decoder block is the SP_decore.lib library.

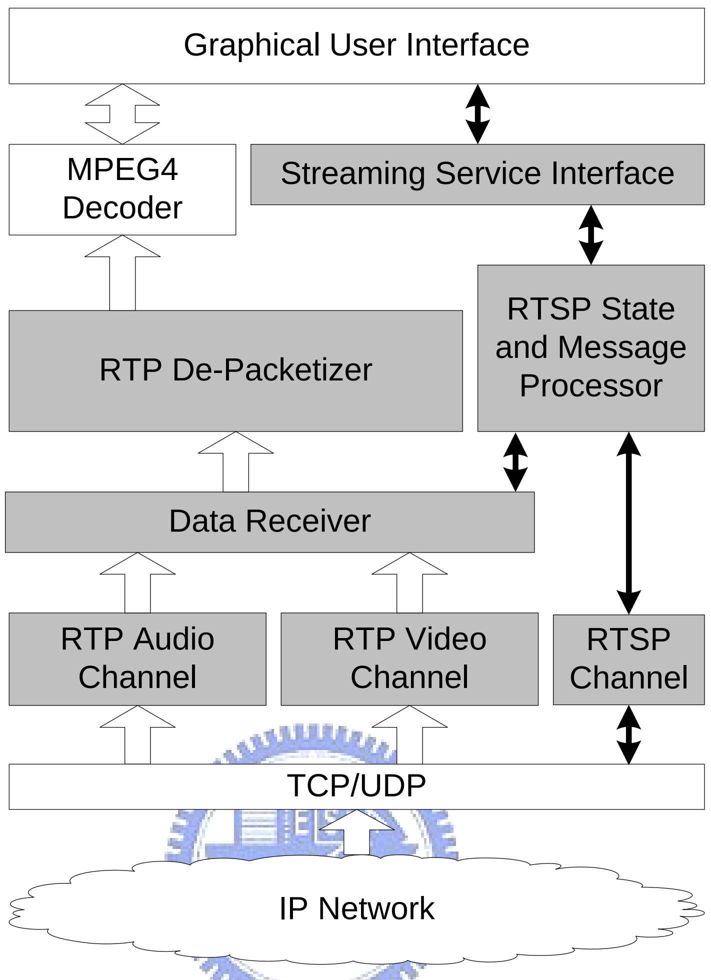


Figure 28 – Architecture of client system

Continuing with Figure 28, the GUI block, as described previously, is responsible for the interactions with the user. Whenever the user presses a button or types in some texts in a blank box, the GUI block must be able to trigger an event properly. At the beginning of a streaming session, the user will ask the streaming service interface to setup the connection with the server and check the availability of the desired media file in the server's storage by clicking the open or play button. The streaming service interface block in turn will call the RTSP state and message processor to handle the user's command by exchanging RTSP control messages with the server via the RTSP channel. When the data transmission begins, the data receiver, who receives RTP packets, will place received packets on the buffer where the RTP de-packetizer block can access and de-packetize those packets. After de-packetizing, original compressed frames will form and are ready to be decoded by the MPEG-4 decoder block. Eventually, the decompressed frames will be playbacked (displayed) on the GUI.

4.3.2.1 Threads Used in Streaming Client System

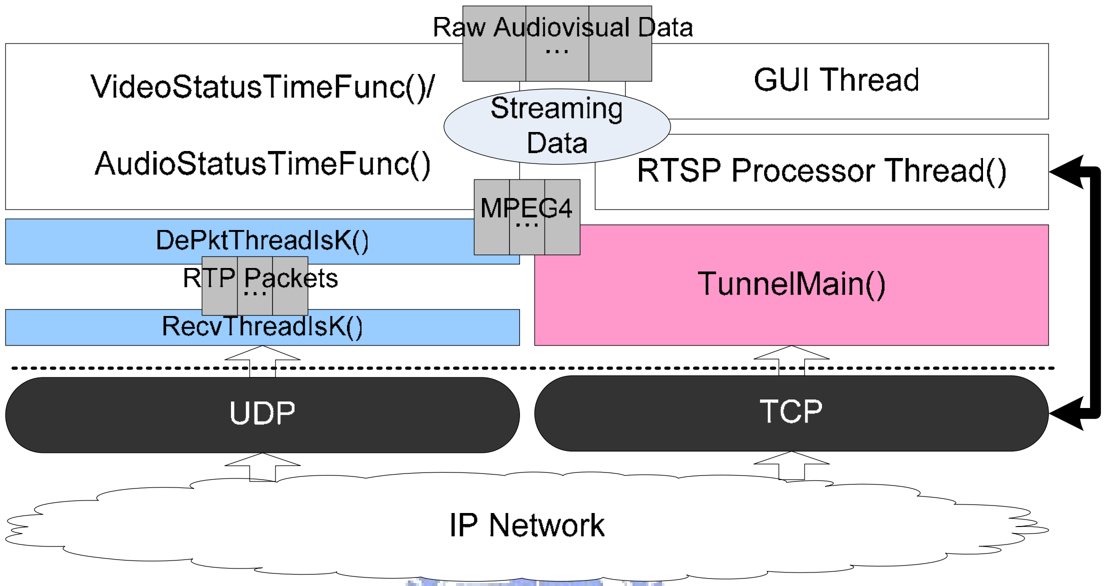


Figure 29 – Threads of client system

After discussing about the tasks and the architecture of the client system, it is time to explain how the system can be implemented in software. Apparently, since lots of tasks have to be handled at the same time by the client program, multiple threads are definitely required.

Referring to Figure 29, there are several threads running simultaneously in the client system when the RTSP session is in progress. Note that the overlaps between the gray ellipses/rectangles and threads (white rectangles) represent the relationships between the shared data (gray shapes) and the threads. Specifically, the following threads will execute once the client starts:

- GUI thread
  - Responsible for interacting with the user of the client program.
- *VideoStatusTimeFunc()*
  - Responsible for collecting and displaying the video receiving status on the GUI. In addition, it is also responsible for getting received MPEG-4 video frames from the buffer where the receiving threads can access, and displaying them on the GUI.

- *AudioStatusTimeFunc()*
  - Responsible for collecting and displaying the audio receiving status on the GUI. In addition, it is also responsible for getting received MPEG-4 audio frames from the buffer where the receiving threads can access, and playback them.
- *RTSPProcessorThread()*
  - Responsible for sending, accepting, parsing, and extracting the RTSP messages, and performing necessary state transitions. It is equivalent to the session controller shown in Figure 25.

The following thread is created only when TCP is selected to be the means of media data transmission.

- *TunnelMain()*
  - Responsible for receiving the streamed media data frames via TCP, and placing them on the buffer where *"AudioStatusTimeFunc()"* or *"VideoStatusTimeFunc()"* can access.

However, the following two types of threads are little bit different. In fact, there will be a pair of instances for each type. One is for the video connection, while the other is for the audio connection. In addition, they are created only when UDP is selected to be the means of media data transmission.

- *RecvThreadIsK()*
  - Responsible for receiving the streamed RTP packets via UDP, and placing them on the buffer where "*DePktThreadIsK()*" can access.
- *DePktThreadIsK()*
  - Responsible for de-packetizing the RTP packets received by "*RecvThreadIsK()*" thread, and placing them on the buffer where *"AudioStatusTimeFunc()"* or *"VideoStatusTimeFunc()"* can access.

4.4 Graphical User Interface Thread

For a system to be interactive with the user, it must have an interface where the user and it can communicate with each other. To be more user-friendly, nowadays a type of interface called the graphical user interface (GUI) is commonly used. In the proposed system, GUI plays an important role who allows the user and the system to interact with each other. In a GUI, usually there is more than one control item present. For example, there might be a button to start and a button to stop some service. For another example, if the system is a media player, then there should be an item (picture box) that can render the images on the monitor so the user can view them. For the proposed system, there is a GUI for the server and another for the client. The required control items for each of both are explained in the following section.

4.4.1 Control Items Required

The following list defines a minimal set of control items that the server's GUI must contain in order to provide the desired streaming functions. When one of the buttons is triggered (i.e. pressed), one or more than one interface function contained in the streaming service interface block will be called to handle this event. The streaming service interface blocks for the server and the client will be introduced and explained in Section 5.3 and Section 5.4 respectively.

- A "Start" button
  - This button allows the user to start streaming service.
- A "Stop" button
  - This button allows the user to stop streaming service.
- A "Preview" button for each connected USB camera
  - This button allows the user to activate the USB camera and preview the images that the camera captures.
- A "Stop Previewing" button for each connected USB camera
  - This button allows the user to deactivate the USB camera and stop previewing.
- An "URI" edit field for each IP wireless camera
  - This edit field is the place where the user can enter the IP address of the wireless camera.
- A "Connect" button for each IP wireless camera
  - This button allows the user to connect to the wireless camera and preview the images that the camera captures.
- A "Stop Connecting" button for each IP wireless camera

- This button allows the user to disconnect the wireless camera and stop previewing.

The following list defines a minimal set of control items that the client's GUI must contain in order to provide the desired streaming functions.

- An "URI" edit field
  - This edit field is the place where the user can enter the IP address and the name of the desired media file.
- An "Open" button
  - This button allows the user to connect to the streaming server.
- A "Stop" button
  - This button allows the user to disconnect with the streaming server.
- A "Pause" button
  - This button allows the user to pause
- A "Seek to" button
  - This button allows the user to forward or backward the presentation to a specified time instant.
- An "Seek Time" edit field
  - This edit field is the place where the user can enter the desired time instant that the presentation should be sought to.
- A "TCP Tunnel" check box
  - This check box allows the user to specify whether to receive RTP packets via TCP or UDP channel.

In fact, the reason why the streaming service blocks on both the server and the client can be bridges connecting the GUIs and the underlying modules is that because whenever the user triggers an event and a corresponding function in the streaming service interface block is called to handle this event, the corresponding member function of the instance of "*RTSPStreamerServer*" or "*RTSPStreamClient*" created for this streaming session will be called to deal with this event. Figure 30 illustrates this idea. In summary, GUI accepts the command from the user and calls the corresponding function provided by the streaming service interface block. The called function in turn will call the corresponding member function of the instance of "*RTSPStreamerServer*" or "*RTSPStreamClient*" created for this streaming session. This member function, finally, will perform the necessary jobs. The following two sections describe the "*RTSPStreamerServer*" or "*RTSPStreamClient*" classes.

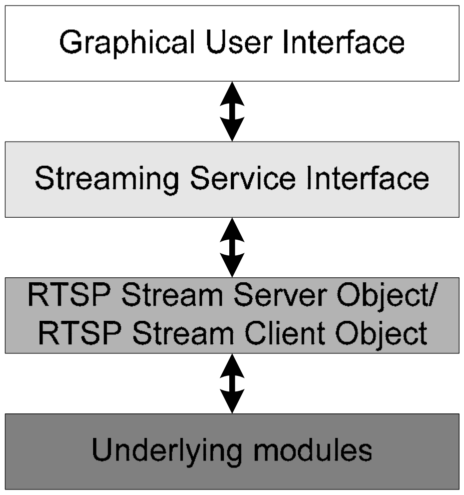

Graphical User Interface
Streaming Service Interface
RTSP Stream Server Object/
RTSP Stream Client Object
Underlying modules

Figure 30 – Relationships between GUI and underlying modules

4.4.2 Members of RTSPStreamServer Class

The following bulleted list describes when an event is triggered, what function in the streaming service interface block will be called. In addition, it also portrays what member function of the instance of "*RTSPStreamerServer*" will in turn be used to handle the event.

- When the "Start" button is pressed (referring to Figure 31)
  - "*StartServer()*" in the streaming service interface block will be called to create an "*RTSPStreamerServer*" object, which handles all the streaming tasks, including RTP and RTSP related matters.
- When the "Stop" button is pressed
  - "*StopServer()*" in the streaming service interface block will be called to delete the created "*RTSPStreamerServer*" object so all the streaming services provided will be canceled.

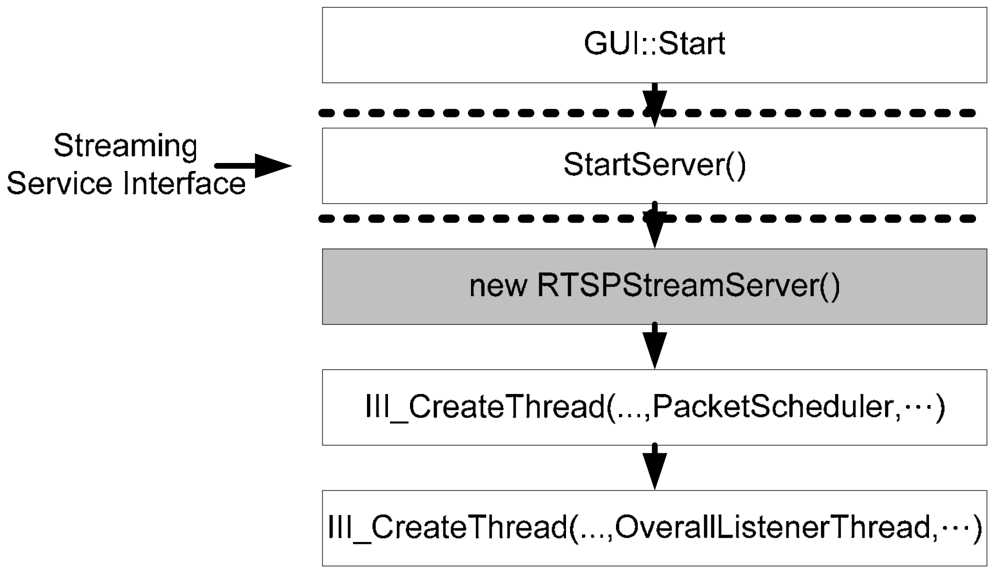

GUI::Start
Streaming Service Interface
StartServer()
new RTSPStreamServer()
III_CreateThread(...,PacketScheduler,⋯)
III_CreateThread(...,OverallListenerThread,⋯)

Figure 31 – When “Start” button is pressed

- When a live-captured media frame is compressed and ready to be placed onto the output buffer
  - “*CaptureToServer()*” in the streaming service interface block will be called.
  - The member function “*SelectCaptureBufferLink()*” in turn will be called to place this frame onto the output buffer where the data packetizer can access.
- When GUI desires to know the number of connected clients and the status of each
  - “*GetClientNumber()*” and “*GetClientStatus()*” of the streaming service interface block will be called.
  - The number of connected clients and the status of each client stored in the “*RTSPStreamerServer*” object will be read by the above two functions, which in turn will return the results to GUI.

4.4.3 Members of RTSPStreamClient Class

The following bulleted list describes when an event is triggered, what function in the streaming service interface block will be called. In addition, it also portrays what member function of the instance of “*RTSPStreamerClient*” will in turn be used to handle the event.

- When the “Open” button is pressed (referring to Figure 32)
  - “*StreamCreateClient()*” in the streaming service interface block will be called to create an “*RTSPStreamClient*” object, which handles all the streaming tasks,

including RTP and RTSP related matters.

- “*StreamOpenConnection()*” in the streaming service interface block will be called. This function in turn will call the member function “*Open()*”, which will connect to server’s RTSP port, create channels to receive streamed data, and send DESCRIBE, SETUP and PLAY requests to the server.

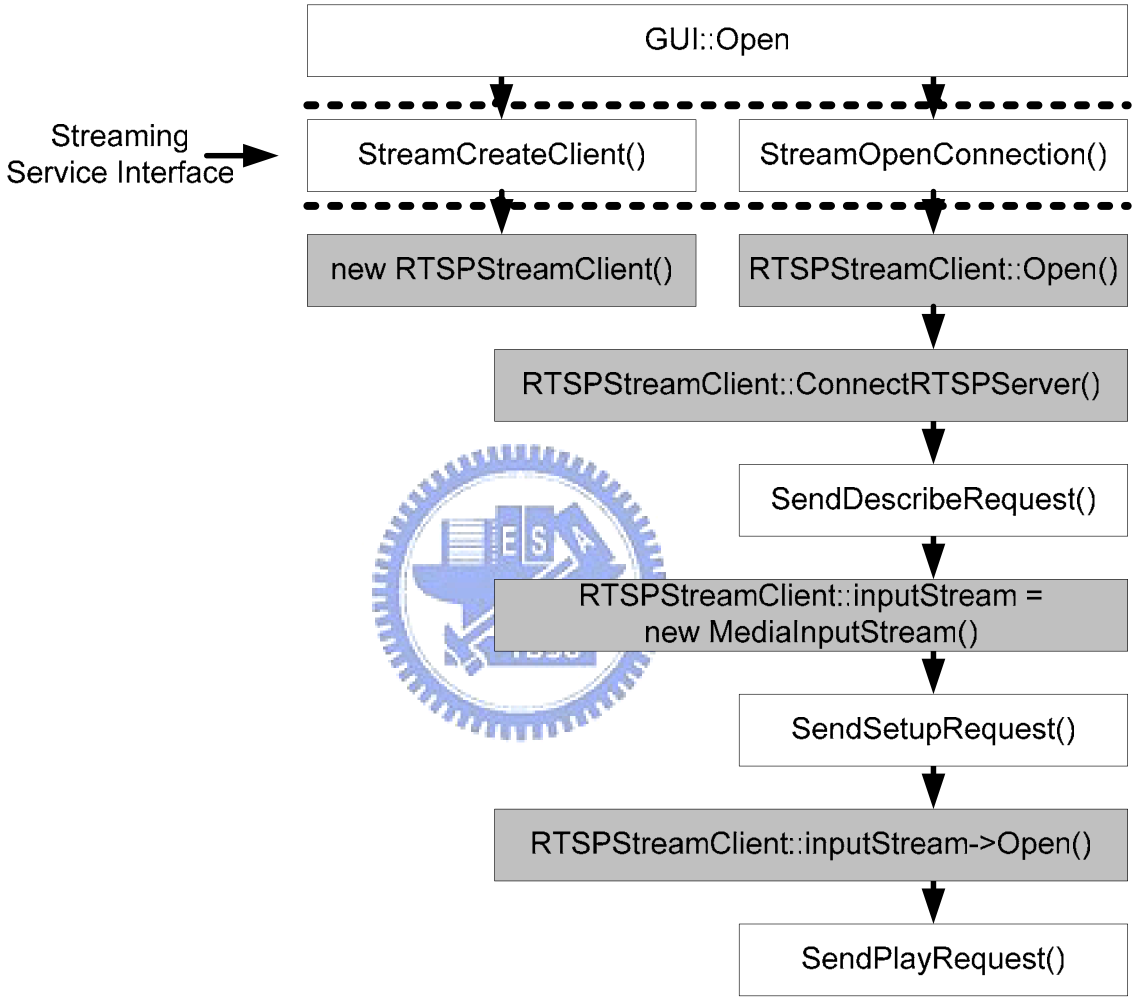

GUI::Open
Streaming Service Interface
StreamCreateClient()
StreamOpenConnection()
new RTSPStreamClient()
RTSPStreamClient::Open()
RTSPStreamClient::ConnectRTSPServer()
SendDescribeRequest()
RTSPStreamClient::inputStream = new MediaInputStream()
SendSetupRequest()
RTSPStreamClient::inputStream->Open()
SendPlayRequest()

Figure 32 – When “Open” button is pressed

- When the “Stop” button is pressed
  - “*StreamCloseConnection()*” in the streaming service interface block will be called. It in turn will call the member function “*Close()*” to close the connection with the server.
  - “*StreamReleaseClient ()*” in the streaming service interface block will be called to release all the resources associated with the “*RTSPStreamerClient*” object and then delete this object.

- When the "Pause" button is pressed
  - "*StreamPauseConnection()*" in the streaming service interface block will be called. It in turn will call the member function "*SendPauseCmd()*", which will send pause command to the "*RTSPProcessoreThread()*" if the current state is PLAY; otherwise, "*SendPlayCmd()*" will send play command to the "*RTSPProcessoreThread()*" . Details about this thread will be given the later section.
- When the "Seek to" button is pressed
  - "*StreamPlayConnection()*" in the streaming service interface block will be called. It in turn will call the member function "*SendPlayCmd()*", which sends play command to the "*RTSPProcessoreThread()*". Details about this thread will be given the later section.
  - After the presentation is resumed, "*StreamSeekConnection()*" in the streaming service interface block will be called. It in turn will call the member function "*SeekStream()*", which eventually will make the server transmit the data of the desired presentation time instant.
- To get a received frame
  - "*VideoStatusTimeFunc()*" or "*AudioStatusTimeFunc()*" can call "*StreamGetVideoFrame()*" or "*StreamGetAudioFrame()*" in the streaming service interface block, respectively, which in turn will call the member function "*GetVideoFrame()*" or "*GetAudioFrame()*". These sequential calls will result in returning a media frame to the thread, which upon receiving the frame, can use the decoder to decompress it and ask the GUI to render it.
- To know the current presentation time
  - "*VideoStatusTimeFunc()*" or "*AudioStatusTimeFunc()*" can call "*StreamGetCurrentVideoTime()*" or "*StreamGetAudioFrame()*" in the streaming service interface block, respectively, which in turn will call the member function "*GetCurrentVideoTime()*" or "*GetCurrentAudioTime()*". These sequential calls will result in returning the timestamp of the current frame to the thread, which can ask the GUI to display this information to the user.
- To find out the length of the duration that the received frames still residing on the buffer but not yet rendered on the display device are worthy of

- "*VideoStatusTimeFunc()*" or "*AudioStatusTimeFunc()*" can call "*StreamGetBufferVideoTime()*" or "*StreamGetBufferAudioTime ()*" in the streaming service interface block, respectively, which in turn will call the member function "*GetBufferVideoTime()*" or "*GetBufferAudioTime()*". These sequential calls will result in returning the buffer length to the thread, which can ask the GUI to display this information to the user. .

- To find out the current RTSP state
  - "StreamGetState*()*" in the streaming service interface block will be called.
  - The current RTSP state stored in the "*RTSPStreamerServer*" object will be read by the function, which in turn will return the results to the calling thread.

4.4.4 Transcoder Thread

This thread is only present on the server side and is created when the streaming service is up and running. Essentially, it plays a very important role of transcoding live-captured media data to the desired format. More precisely, it is responsible for acquiring audiovisual contents from various types of cameras, encoding or transcoding them into MPEG-4 format, and placing the resulted bitstreams onto the buffer where "*StreamerletIs()*" thread can access.

Stepping a bit deeper, before this thread kicks off, the GUI has to create MPEG-4 virtual files by getting help from 3GP creator module first. The reason of doing so is simple: when a client wishing to view the live-captured audiovisual contents connects to the server, the client has to see (feel) that there are MPEG-4 files which are associated with different cameras available in the server storage. Therefore, before transcoding, the GUI has to make these non-exiting files virtually exist in the database. After this step is done, "*Transcoder()*" thread can get started. This thread does the following tasks in a loop:

- It uses the API provided by wireless cameras' SDK or Viscom Software VideoCap Live v1.5 ActiveX Control to acquire frames from various types of cameras.
- If the frames are from the selected wireless cameras, these frames are in MJPEG format. Therefore, they have to be decoded by using MJPEG decoder, which is included in the SDK as well. If the frames are from USB cameras, they are raw data.

Thus, no decoding is required.

- Now the captured data are in raw format, the thread can render them on the display device if desired.
- The thread will feed these raw data into MPEG-4 encoder.
- The thread will call “*CaptureToServer()*” described earlier to place each MPEG-4 frame onto the output buffer.

As a side note, the frequency that this loop cycles has to match the desired frame rate as not to acquire too many or too few frames from the cameras. To implement this, every time the thread reaches the end of the look, it has to check the system time. If it is still too early to cycle again, this thread will call “*Sleep(1)*” to sleep for one millisecond. When waking up, the thread will check again and sleep again if necessary. This job is repeated until it is time to run the loop again.

4.4.5 StatusTimeFunc

This thread is only present on the server side and is created when the streaming service is up and running. It is responsible for collecting and displaying all the clients’ statuses on the GUI. Because the streaming service interface block is designed in such a way that the GUI can communicate with the underlying modules effortlessly, this thread function is very simple to implement. Essentially, as described in the earlier section, it will call “GetClientNumber()” to know how many clients are connecting to it. With this number, it will know how many times “*GetClientStatus()*” should be called to get each client’s status. These statuses can be displayed on the display device by simply using “*SetWindowText()*” (function call provided by Windows®) to place texts on the desired edit field.

4.4.6 VideoStatusTimeFunc and AudioStatusTimeFunc

These two threads are only present on the client side when the streaming is in progress. They perform identical tasks except that one is for the video and the other is for audio. To make this section easier to follow and understand thoroughly, only “*VideoStatusTimeFunc()*” will be discussed here. Note that this thread is where the image processing functions should be inserted because, as described in earlier section, this thread is responsible for getting received MPEG-4 video frames from the receiving buffer, and displaying them on the GUI.

Naturally, this thread runs in a loop of getting a frame from receiving buffer, decoding it, and displaying it on the GUI. Initially, it will call "*StreamGetBufferVideoTime()*" to find out how many frames are still in the receiving buffer (i.e. have not been rendered). This number of frames may be expressed in timestamp units to represent the buffer length. If the length of the buffer is too short (i.e. shorter than the pre-defined threshold), it will sleep for a while and start the loop again; otherwise, it will call "*StreamGetVideoFrame()*" to get a MPEG-4 compressed frame. To be able to display the frame on the GUI, this thread has to use MPEG-4 decoder to decompress the compressed data. Thus, it uses the function "*decore()*" provided by SP_decore.lib library to decode the frame. Now, since the frame is in the uncompressed format (raw data), it is time for the image processing functions to kick off. The image processing function can be implemented in two different ways: a normal subroutine function call or an independent thread. The choice can be made based on the application needs and the complexity of the image processing algorithm. For example, if the image processing algorithm does not consume too much computational power (i.e. easy and fast to finish the processing), it can be implemented as a subroutine call. Therefore, after the decoding process is done, the thread will call this image processing function. The instructions below this function call will not executed until the function returns. This is okay since the execution time of this function will not be too long. In other words, the loop that this thread is running will not be held by the image processing function. On the other hand, if the image processing algorithm is a heavy-weighted one, it cannot be implemented as a function call, since it will take a long time to process the image. Consequently, it has to be run in a separate thread. In short, "*VideoStatusTimeFunc()*", after decoding each frame, can place it on a temporary buffer. Then, this independent image processing function can always take a frame from this temporary buffer to perform its analysis, and report to the GUI when detecting an abnormal event.

At the end of the loop, the status about this streaming can be acquired by using functions provided by the streaming service interface block such as

- *StreamGetCurrentVideoTime()*
- *StreamGetBufferVideoTime()*
- *StreamGetTimeScale()*
- *StreamGetLossRate()*
- *StreamGetTransmitRate()*

and displayed on the GUI.

4.5 Implementation of RTSP Messages and State Changes

On the server side, for each connection successfully established between the server and a client, the "*OverallListenerThread()*" will create another thread named "*RTSPProcessorThread()*", whose job is to handle all the exchanges of RTSP messages between the server and the client. These messages include RTSP requests sent by the client and the RTSP responses replied by the server. For each incoming RTSP request, "*RTSPProcessorThread()*" will first check its validity. If the request is invalid, it will be discarded; otherwise, another function called "*handle_server_event()*" will be utilized to perform the proper state change if necessary. For example, if the client sends a PLAY request to the server, who is currently in READY state, "*handle_server_event()*" will realize that the client has demanded to start the streaming process, and it is time to transit to PLAY state. On the other hand, the packetizer thread (named "*StreamerletIs()*" in this implementation), which has been waiting for the PLAY request and is currently being halted, will see that the RTSP state is changed to PLAY state, so it will start the streaming process. How "*RTSPProcessorThread()*" and "*StreamerletIs()*" can communicate with each other is discussed below. The basic way to implement the inter-thread communication is through the use of a shared memory. To avoid conflicts that occur when several threads are accessing the same memory, the notion of mutex is utilized. Details about mutex are given in previous section.

Before looking into how the inter-thread communication is carried out, two structures have to be described first. The first structure is "*STREAMING_DATA*":

```
typedef struct _Streaming_data{
    plll_mutex hRTSP_STREAMER_Mutex;
    RTSP_STREAMER controlData;
    …
}STREAMING_DATA;
```

Note that a lot of variables contained in this structure are omitted for simplicity. This structure contains a mutex named "*hRTSP_STREAMER_Mutex*" and "*controlData*", which is the second structure to be introduced:

```
typedef struct _RTSP_STREAMER{
    int    cmdis;
    int    states;
    …
}RTSP_STREAMER;
```

This structure contains two integers, "*cmdis*" and "*states*". Unsurprisingly, "*STREAMING_DATA*" is the shared memory shared by the two threads, and "*hRTSP_STREAMER_Mutex*" is used to protect this shared memory from being used by the two simultaneously. In addition, "*cmdis*" represents the latest received RTSP request and "*states*" is the current RTSP state. For the same example mentioned above, in "*handle_server_event()*", "*RTSPProcessorThread()*" will lock "*hRTSP_STREAMER_Mutex*" first, avoiding "*StreamerletIs()*" to use the "*STREAMING_DATA*" at the same time. After acquiring the mutex, "*RTSPProcessorThread()*" will change "*cmdis*" to the PLAY request and "*states*" from READY to the PLAY state. "*StreamerletIs()*", upon sensing this change by reading the shared memory (mutual exclusion guaranteed by the mutex), will start the packetization process.

4.5.1 OverallListenerThread

According to RTSP [3], RTSP typically runs over a reliable connection based transport level protocol such as TCP. Therefore, to be more compliant with other streaming solutions, in the proposed system, TCP is chosen to be the channel for the RTSP communication between the client and the server. Since TCP is connection-oriented (i.e. the connection between two ends has to be setup before the actual transmission takes place), the setup procedures must be taken care of. This is the main job of *"OverallListenerThread()"*. It assists the server to accept an incoming connection attempt of a client on the designated socket. After this connection is setup successfully, it passes the control of this connection to another thread whose job is to receive and reply RTSP messages, and starts to wait for another new incoming connection.

In the proposed system, on the server side, the socket bound to port 554 is specially opened to listen for an incoming connection from the client. To bind this socket with port 554 and let it start listening, the following code is executed:

```
struct sockaddr_in s_in;
int fd = socket(AF_INET, SOCK_STREAM,IPPROTO_TCP);
s_in.sin_family = AF_INET;
s_in.sin_addr.s_addr = INADDR_ANY;
s_in.sin_port = htons(554);
bind(fd, &s_in, sizeof(struct sockaddr));
listen(fd, 5);
```

The second line creates a socket file descriptor named *"fd"*, which is used to handle a socket, and binds it to a specific transport service provider. In this case, the first parameter is the address family specification and set *"AF_INET"* by default. The second and the third parameters indicate that this socket is a reliable two-way connected communication stream (TCP). Now it is time to bind this socket to a port on the local machine. As can be observed from the code, *"s_in"* is a pointer to a structure named *"sockaddr_in"* that contains information about the port and IP address. *"INADDR_ANY"* denotes the IP address of this machine. The port, as desired, is set to 554. The second last line then binds *"fd"* to port 554 using *"s_in"*. The last line, finally, makes *"fd"* socket listen for an incoming connection.

After letting the *"fd"* start listening on port 554, *"OverallListenerThread()"* can begin its job. The following few lines summarize the job that this thread is working on:

```
while(server is running){
    struct sockaddr_in client_addr;
    int sin_size = sizeof(struct sockaddr_in);
    int client_fd = accept(fd,&client_addr,&sin_size);
    /*Creates an instance of RTSPProcessorThread() to handle this established
    connection (i.e. passing client_fd to RTSPProcessorThread()).*/
}
```

Basically, as long as the server is up and running, this thread is listening on port 554 to wait for a new incoming connection from the client. After a connection is accepted, a new socket file descriptor (named *"client_fd"*) will be created to take over the control of this connection (instead of the original *"fd"*). The original socket file descriptor will still be listening on port 554 and will accept the next new incoming connection by calling *"accept()"* function call again. This is done by placing these lines in a while loop. On the other hand, the newly created one will be passed as a parameter to *"RTSPProcessorThread()"*, who will use function calls like *"send()"* and *"recv()"* to perform RTSP message exchanges. According to [20], in this case, *"fd"* are *"client_fd"* are called "listening socket" and "connected socket", respectively, since *"fd"* during its life time, always listens to new incoming connection, while *"client_fd"*, on the other hand is created and used to perform actual data transmission when a connection is successfully setup.

As a side note, most of TCP socket related structures and functions employed in the implementation, such as *"struct sockaddr_in"*, *"socket()"*, *"bind()"*, *"connect()"*, *"listen()"*, *"accept()"*, *"send()"*, and *"recv()"*, are contained in the Windows® Socket library. They can be used directly after including wsock32.lib and winsock.h in the program.

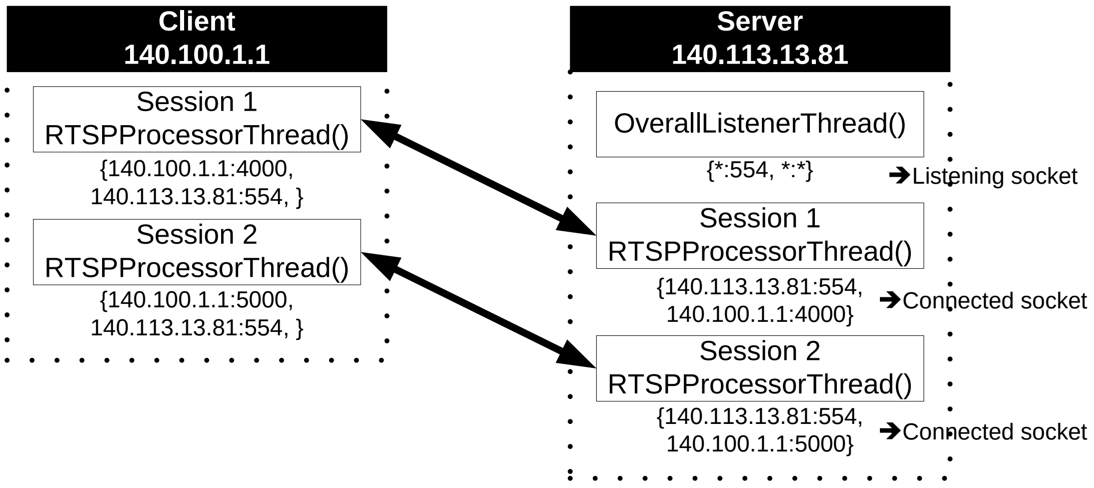


Figure 33 – Illustration of multiple RTSP sessions connecting to a server [20]

Figure 33 [20] is a typical scenario when a client creates multiple streaming sessions with a single server. It illustrates how the server deals with multiple socket file descriptors bound to the same port (554). Initially, on the server side, *"OverallListenerThread()"* will call *"socket()"*, *"bind()"*, and *"listen()"* to create a socket that listens to incoming connection from the client. As mentioned above, this socket is called "listening socket" [20]. At this time, since no connection has been setup on port 554, any incoming connection attempt with any source/destination pair will be handled by this socket without a doubt. When the client on the left hand side creates Session 1 and calls *"connect()"* to connect to the server, the server will call *"accept()"* to accept this connection. *"accept()"* function, upon executing successfully, will return a new socket handler to take over the control of this connection. So this new socket is called "connected socket" [20], as described earlier. Therefore, on the server side, both the original socket and the newly created socket are now bound to port 554. If again, the client creates another session and connects to the server via port 554 on the server machine, then the server will have the original socket, and two newly created sockets bound to port 554. How the server distinguishes the packets sent from Session 1 and Session 2 which are of same IP address, and how it differentiates the data packets and connecting requests from new clients, are described below. Note that the wildcard notation within the braces beneath *"OverallListenerThread()"* denotes that the listening socket listens to any incoming connection with any source address other than ones already managed by other sockets. Therefore, if a packet from 140.100.1.1:4000 and destined to 140.113.13.81:554 will not be handled by this listening socket. Indeed, it will be captured by the connected socket of Session 1 since its source/destination pair matches that of this socket. In summary, even though multiple socket file descriptors are bound to 554, any incoming packet or connecting request will not be captured by the wrong socket since source/destination pair information contained in the packet will enable the underlying kernel program to deliver the packet to the correct socket.

4.5.2 RTSPProcessorThread

This thread is implemented on both the server and the client systems. Its main tasks are to handle RTSP message exchanges between the client and the server, and call corresponding event handling functions.

In the thread, the system keeps on waiting for incoming RTSP messages. Because multiple RTSP messages may be bundled to be transmitted in a single TCP packet, the function call "*extract_one_rtsp()*" is continuously utilized to extract a single RTSP message from the bundled RTSP messages contained in a received TCP packet until all RTSP messages are processed. Each time an RTSP message is extracted, the subroutine "*ParseRTSP()*" is called to check whether it is a response or request message. If it is a request message, the RTSP method contained in it will be read out. After described functions are executed successfully, the part that differs between the server and the client begins. In particular, the server will call "*handle_server_event()*" and the client will call "*handle_client_event()*". Both functions are to perform the necessary event handling tasks and RTSP state transitions.

Before the two handlers are explained in detail, one thing that worth noting has to be mentioned. Recall that it is essential that both the server and the client have to send HELLO requests and reply HELLO response to each other on a periodic basis. In this implementation, the "*RTSPProcessorThread()*" of the client will send HELLO request to the server periodically by using "*send_hello()*" function call. Upon receiving the HELLO request, in addition to replying with an OK response (code 200 in Table 2) by using "*send_reply()*" to the client, the server will send another HELLO request to the client, in the hope of that the client will also reply an OK response. In this way, both the server and the client will periodically send a HELLO request to each other. If nothing goes wrong, they will reply to each other with an OK response, as specified by RFC 2326 [3].

4.5.2.1 handle_server_event

This function is implemented in a double-nested manner. That is, inside a structure of switch and case statements, there is another switch-case structure. On the outer switch-case structure, the expression being evaluated is the current RTSP state, while for the inner one, the expression being evaluated is the RTSP operational code contained in the RTSP message. This function is done in this way because obviously, for the RTSP state machine to transit from one state to another state correctly, and for the server to perform appropriate event handling tasks, the current RTSP state has to be known before anything else takes place. After recognizing the current RTSP state, the RTSP operational code (i.e. RTSP method or response) has to be examined. If its value is invalid for the current state, for example, the server receives PLAY or PAUSE request while in Init state, then

```
send_reply(455,…, "Accept: OPTIONS, DESCRIBE, SETUP, TEARDOWN\n",…);
```

is called to send an error message to the client saying that this method is not valid for the current state and informing the client about all the valid requests. In addition, if the server cannot recognize this operational code, then

```
send_reply(501, …, "Accept: OPTIONS, DESCRIBE, SETUP, TEARDOWN\n",…)
```

is called to tell the client that this operational code is not implemented. If the operational code is valid for the current state, then an event handling function will be called. For example, when the current state is Init and SETUP request is received and recognized, then the following tasks will be done by the event handler "handle_setup_request()".

- It will check if the number of the connected clients has reached the maximum value. If it has, then the server will call "*send_reply(453,…)*" to inform the client that there is not enough bandwidth.
- It will parse the setup information contained in this RTSP request message so the desired settings can be applied to the server's transport system. For example, in the request message the client might specify whether the UDP or TCP should be employed to deliver the RTP packets. For another example, the client will specify which track in the media file that it wants the server to setup.
- The most important of all, it will call "stream_server_start()", which in turn will create "*StreamerletIs()"* thread to perform packetization task.
- Finally, after the above jobs have been successfully accomplished, it will call "*send_setup_reply ()*" to inform the client that the setup request has been fulfilled, and about the newly created session ID.

4.5.2.2 handle_client_event

This function works in a similar way as "*handle_server_event()*". However, unlike "*handle_server_event()*" which deals with RTSP requests most of the time, it handles RTSP responses. Unsurprisingly, the client will make a state transition only when it receives an OK (status code = 200) response sent by the server. For example, after the client sends a PLAY request to the server while in Ready state, it will not move to Play state right away. Instead, it will transit to Play state after receiving the OK response indicating that the PLAY request has been granted. Now two questions arise. Who generates the RTSP requests and who sends them to the server? These questions can be solved easily by observing Figure 30. Remember that GUI and the underlying services (e.g. "*handle_client_event()*") can communicate with each other via the streaming service interface block and "*RTSPStreamerClient*" object. Therefore, after the user presses the "Pause" button in the GUI, for example, "*StreamPauseConnection()*" in the streaming service interface block will be called by the GUI event handling function. "*StreamPauseConnection()*" will in turn call "*SendPauseCmd()*", which is a member function "*RTSPStreamerClient*". "*SendPauseCmd()*" will call "*SendPauseRequest()*", which finally will call "*handle_client_event()*" and give it the CMD_PAUSE command. "*handle_client_event()*", like usual, will evaluate two expressions as described earlier: the current RTSP state and the operational code. In this case, it will realize that the current state is Play state and the operational code is CMD_PAUSE. Thus, it will call "*send_pause_request()*" to actually send the RTSP PAUSE request to the server.

4.6 Implementation of RTP Streaming

In this section, how the RTP transmission channels are setup, and the packetizer, de-packetizer, streaming control, and bandwidth smoothing are implemented in the software will be explained exhaustively.

4.6.1 RTP Transmission Channel Setup

Without a doubt, RTP transmission channels have to be setup before the actual streaming process can kick off. Thus, it is essential that the operations involved for the proposed client/server systems to establish the RTP channels must be introduced before discussing about the RTP streaming.

In fact, the data transmission channels are setup after client and server have exchanged DESCRIBE request and response as required by RTSP [3]. For the server to know at which ports the client is waiting for the media data, there must be means of communication. In effect, this is carried out by the exchanges of SETUP messages. For simplicity, now it is assumed that the targeted media data contain only video; thus, only one port for the client and another for the server are required to transmit the media data. First of all, the client has to find an unoccupied port randomly (let's say port X). In SETUP request sent by the client to the server, the client will contain the information about port X and whether the data is to be transmitted over TCP or UDP. If the TCP is chosen, the following procedure will be followed:

1. The client will create a socket that in turn will try to connect to port X on the server side.
2. Upon receiving information about port X, the server will create a socket that in turn will listen and accept an incoming connection on port X.
3. Obviously, the above two steps will result in establishing a connection between the client and the server. The server will create "*StreamerletIs()*", which feeds media frames to the data streamer (i.e. "*TunnelMain()*" in this case). Sequentially, both the client and the server will create their own "*TunnelMain()*" threads and are now ready for the streaming process.
4. The server will reply SETUP response to the client indicating that SETUP request was granted.

On the other hand, if UDP is chosen, the following procedure will be followed:

1. The client will create "*RecvThreadIsK()*", which contains a socket that receives RTP packets on port X.
2. Upon receiving information about port X, the server will create "*StreamerletIs()*", which feeds media frames to the data streamer (i.e. "*PacketScheduler()*" in this case) along with destination IP address and port (i.e. port X). As a result, each time "*PacketScheduler()*" receives a frame from "*StreamerletIs()*", it will know to what IP address and what port this packet should be sent to.
3. The server will reply SETUP response to the client indicating that SETUP request was granted.

4.6.2 Packetizer and De-Packetizer

For the server system, a structure named “rtp_hdr_t”, which represents the RTP header as defined in RFC 3550 [2], is defined as follows:

```
typedef struct {
    unsigned int version:2;
    unsigned int p:1;
    unsigned int x:1;
    unsigned int cc:4;
    unsigned int m:1;
    unsigned int pt:7;
    unsigned int seq:16;
    u_int32 ts;
    u_int32 ssrc;
    u_int32 csrc[1];
} rtp_hdr_t;
```

Observing from the above code, this structure is compliant with RFC 3550 [2]. It contains two bits for the protocol version, one bit for the padding bit, one bit for the extension bit, four bits for the CSRC count, one bit for the marker bit, seven bits for the payload type, two bytes for the sequence number, four bytes for the timestamp, four bytes for the SSRC, and 4 bytes for each of the CSRC. The purposes and the usages of these fields are described in Section 3.1.2.1.

How the packetizer works in the server system is as follows. It takes one frame from the media file at a time, divides it into fixed-sized segments (usually 1024 bytes), and adds an RTP header structure described above to each of them. Certainly, all the bit fields in the header structure have to be filled with correct information, in accordance with RFC 3550 [2]. Moreover, in the proposed system, following this standard header, three extra fields are piggybacked:

- int ChunkSize;
- int flags_tot;
- int offset;

“*ChunckSize*” is the summation of the size of these three fields and the size of the payload. “flags_tot” is to indicate whether this packet is a subpart of a key frame or a predicted frame. With this information, the data streamer can decide whether to drop delayed packets or not. If a packet is delayed but it is a subpart of a key frame, then it will not be dropped by the data streamer. On the other hand, if the data streamer is about to transmit a packet belonging to a predicted frame, but finds it delayed intolerably, the data streamer may drop it, if desired, to reduce the level of congestion. The field “offset” represents the difference between the position of the starting byte of this frame and that of the payload contained in this packet. With this information, the de-packetizer can find out what part of the frame this packet belongs to.

On the client side, the de-packetizer is present. As implied by its name, the de-packetizer in effect reverses the work done by the packetizer. Whenever it receives an RTP packet, it will check that whether the SSRC contained in the header of the packet agrees with that of this RTP session. If they don’t match, this packet will be dropped; otherwise, the packet will be analyzed and processed further. In particular, its sequence number and timestamp will be check against those of previously received RTP packets. For example, if an incoming packet’s sequence number is smaller than that of any previously received packet (i.e. normally, its sequence number should be larger than those of received packets), the de-packetizer will try to solve this problem by reordering the receiving buffer according to the received packets’ sequence numbers. For another example, if some packets belonging to a frame that is about to be playbacked are seriously delayed, the de-packetizer cannot wait any longer and must hand over this incomplete frame to the decoder. Eventually, these delayed packets will arrive and the de-packetizer will find that their timestamps indeed belong to a frame that has already been rendered. Thus, the de-packetizer will not attempt to reorder the sequence but discard them directly.

4.6.3 Streaming Control

In a client/server system, to allow smooth playback of the media data at the client side, the client’s buffer should never be empty or over-loaded. Therefore, the server must always transmit media data to the client according to the schedule defined for the media data as not to drain nor overflow the client’s buffer. Intuitively, the server could use a stopwatch to find out when it should transmit each part of the data. Specifically, the server can record the time it starts the streaming process, and by continuously comparing the current time and the start time, it will know whether it should transmit the next part of the data to the client now or later. In addition, to save the CPU consumption, each time when the server reads its stopwatch and finds that it is still too early to transmit the next part of the media data, it can relinquish the CPU by invoking the sleep request and let other processes run. However, the control of the streaming becomes more complicated if the client is allow to pause, rewind, or forward the presentation of the media data. This chapter gives details about how this complication can be simplified, and how the actual streaming process can be implemented by just using a few variables.

4.6.3.1 Implementation of Streaming Control

In the following paragraphs, timestamps, timescales, pre-sent data, and computations of the start time and sleep time are described.

As a side note, without the loss of generality, the propagation delay and the media decompression time are neglected in the proposed system. In other words, it is assumed that as soon as the data are injected into the network by the server, the client can playback them.

Timestamps and Timescales

Different media files may have different sets of timestamps which advance at different rates, depending on the desired accuracy [2]. To be able to synchronize different streams, for example, an audio track and a video track in a presentation, different timestamps must be able to be converted from one to another, and into the normal clock (i.e. hours, minutes, seconds and milliseconds). This is the use of timescales. To illustrate the meaning of timescales, some examples are provided. For the frame data with the timestamp equal to half of the value of its corresponding timescale will have a presentation time equivalent to 500 milliseconds, while for the frame data with the timestamp equal to a quarter of the value of its corresponding timescale will have a presentation time equivalent to 250 milliseconds. For a typical example, in the proposed system, the timescale for the live-captured video is 1000 while the timescale for the stored video is 90000. To synchronize the two, their least common multiple (i.e. 90000) can be used. Hence, the frame from the former with the timestamp equal to 500 will be played at the same time as the frame from the later with the timestamp equal to 4500, because for the former, 500 over 1000 can be converted into 4500 over 90000, which is the same as that of the later.

Pre-sent data

Since the network jitter and the roundtrip delay changes do exist in the current model of Internet, it is often desirable for the server to transmit data ahead of their actual presentation times so that even when the network congests for a short while, the client can still playback the media data smoothly. The size of the pre-sent data should be made to be smaller or equal to the size of the client buffer as not to overflow the client buffer. In fact, in the proposed system, data are transmitted one second ahead of their actual presentation times. For example, when the streaming process just begins, the frame data with presentation times from 0 to 1 second will be transmitted as fast as possible. After 40 milliseconds, for media with the framerate equal to 25 frames per second, the frame data with presentation time at 1 second and 40 milliseconds will be sent. After another 40 milliseconds, the frame data with presentation time at 1 second and 80 milliseconds will be sent. This process will be continued until the presentation ends. Therefore, it is essential that the system designer uses two variables to keep track of the timestamp of the last transmitted frame and the timestamp of the frame that the client may playback now. In this way, each time when the server looks at its stopwatch, it can figure out the timestamp of the frame that the client may playback currently by calculating the time difference between the current time and the start time of the streaming process, and then compare this timestamp with the timestamp of the last transmitted frame. If the difference between them is smaller than the value of the media’s timescale (i.e. the client playback buffer may not contain data that are worth one-second), it will transmit the next frame immediately; otherwise, it will sleep for a while.

Calculation of Start Time

During the lifetime of a streaming session, there are three major scenarios that should be addressed:

1. Normal presentation: the streaming process continues to run without any user intervention (PAUSE, REWIND, or FORWARD).
2. Paused presentation: the streaming process is paused and then resumed at the very same time instant of presentation.
3. Sought presentation: The streaming process is paused but then resume at a different time instant of presentation. In other words, the presentation of the media data was rewound or forwarded (sought).

In the first scenario, the server periodically reads its stopwatch, performs the calculation, and makes the transmission decision described above. In the second and the third scenarios, because the streaming process is paused for a period of time and then resumed, the time difference between the recorded start time and the current time cannot be used to calculate the correct timestamp of the frame that the client is playing; consequently, the server cannot make the right transmission decision. This problem can be solved by shifting the original start time to a new time instant that is properly placed a certain distance away from the current time. The calculation for this distance is different for these two scenarios because for the former case, the pre-sent frames are still placed at the client's buffer waiting to be played, while for the later case, the pre-sent frames become useless to the client since the presentation is sought (rewound or forwarded) to a new point of time.

For the second scenario, to work out the time distance away from the current time which the new start time should be placed, one has to realize that since the pre-sent frames are still useful to the client, the state of streaming process should be restored to the one just before pausing. In other word, this time distance, in fact, should be equal to the time difference between the old current time and the original start time just before pausing. Thus, it is not hard to imagine that the timestamp of the frame that the client should be playing now and the timestamp of the last transmitted frame remain the same as the ones just before pausing. Since all the conditions for the transmission decision making remain, the result of it that would have been made just before pausing would be identical to the one now.

In the last scenario, since the pre-sent data are no longer useful to the client even though they still reside in the client's buffer, the calculation to derive the new position of the start time is a little bit different. Instinctively, when the streaming process is resumed, the client is waiting for the media data that it rewound or forwarded to (data at the sought presentation time), so the server has to stream those data right away. To make this happen, the time difference between the current time and the new start time should be set to equal to the sought presentation time; equivalently, the new start time should be placed at a distance equal to the value of sought presentation time away from the current time. In addition, since the server has not transmitted any media data ahead of time, the timestamp of the last transmitted frame is equal to the sought presentation time. As a result, the server will not sleep for a few run and will desperately transmit the media data since it sees that the difference between the timestamp of the frame that the client may be playing now and the timestamp of the frame sent lastly is too small, and does not want to drain the client's buffer.

<u>**Calculation of Sleep Time**</u>

As described previously, when the server reads its stopwatch and realizes that it is still too early to transmit the next part of the media data, the server can relinquish the CPU so other processes or threads can be up and running. So how long the server should sleep? If it sleeps too long, it will not be able to follow the schedule of the media file precisely. To the other extreme, if it only rests for a very short period of time, obviously the overhead required for the frequent context switches between processes or threads would degrade the overall system performance. The rule of thumb is that if it is x milliseconds earlier than the time that the next frame should be transmitted, the server should sleep for max(1 , x - c) milliseconds. c milliseconds fixed offset is needed here because if the context switch needed by the server to regain the CPU is delayed for some reason, the server will not be able to wake up on time. Therefore, it is essential to leave some safety margin.

4.6.3.2 Important Variables, Their Meanings, and Their Initial Values

- *struct _timeb StartLocalTime;*
  - This time variable is used to store the time of the instant when the streaming process commences, and is subject to changes because the client may request to pause, rewind, or forward the streaming process of the media data.
- *struct _timeb NowLocalTime;*
  - This time variable is used to store the current system time.
- *int LocalTimeDiff;*
  - This time variable is used to store the time difference between *StartLocalTime* and *NowLocalTime*.
- *bool bPause = false;*
  - *bPause* is a flag used to indicate whether the streaming process has been paused or not.
- *int LastVideoChunkId=LastAudioChunkId = 0;*
  - These variables are used to store the timestamps of the frames previously acquired from the media file.
- *int LastPauseVideoFrameTimestamp= LastPauseAudioFrameTimestamp=0;*
  - These variables are used to store the timestamps of the frames that will be acquired next from the hard drive when the pause command is issued.
- *int NowPauseVideoFrameTimestamp= NowPauseAudioFrameTimestamp=0;*
  - These variables are used to store the timestamps of the frames that will be acquired next from the media if a play request is issued while streaming process has been paused.
- *int StartVideoChunkId=StartAudioChunkId=0;*
  - These variables are used to store the timestamps of the first frames of the video/audio.
- *int NormalVideoChunkId=NormalAudioChunkId=0;*
  - These variables are used to store the timestamps of the frames that the client may be playing at this moment.
- *int VideoTimeScale= mp4_file_info->pCurrentVideoTrackis->mp4vinfois.TimeScale;*

- This variable is used to store the timescale of the streamed video.

- *int AudioTimeScale= mp4_file_info->pCurrentAudioTrackis->mp4ainfois.TimeScale;*
  - This variable is used to store the timescale of the streamed audio.
- *int SleepTime;*
  - This variable represents the duration that this thread should sleep.
- *long tempTime = 0;*
  - *tempTime* is a temporary variable used to calculate the new streaming start time.

4.6.3.3 Software Implementation to Calculate the Start Time

When the streaming process of the requested media data starts for the first time, the server has to record the start time:

```
_ftime(&StartLocalTime);
```

If the streaming process is paused (whether the client seeks to a new presentation time or not), the server will set *bPause* flag to true indicating that the streaming process has been paused. Additionally, the server will record the timestamp of the next frame acquired from the hard drive. By setting the flag and recording the timestamp, when resuming the streaming process, the server will know that the process was paused before, and whether the process was sought to a new presentation time or not (by comparing the recorded timestamp and the timestamp of the next frame acquired from the hard drive now). The following code is executed when pausing:

```
bPause = true;
IIIGetNextSampleTimeStamp(…, VideoTrackCodeName,
                &LastPauseVideoFrameTimestamp);
IIIGetNextSampleTimeStamp(…, AudioTrackCodeName,
                &LastPauseAudioFrameTimestamp);
```

When the server finds that it should start or re-start the streaming process, it should run the following code segment:

```
01|_ftime(&NowLocalTime);
02|if(bPause==true){  // resume streaming
03|    bPause=false;
04|    IIIGetNextSampleTimeStamp(…, VideoTrackCodeName, &NowPauseVideoFrameTimestamp);
05|   IIIGetNextSampleTimeStamp(…, AudioTrackCodeName, &NowPauseAudioFrameTimestamp);
06|   if( NowPauseVideoFrameTimestamp != LastPauseVideoFrameTimestamp){
07|           LastVideoChunkId = NowPauseVideoFrameTimestamp;
08|           LastAudioChunkId = NowPauseAudioFrameTimestamp;
09|           tempTime = NowLocalTime.time*1000 + NowLocalTime.millitm -
10|                   (long)((float)NowPauseVideoFrameTimestamp/(float)VideoTimeScale*1000);
11|   }
12|   else{
13|           tempTime = NowLocalTime.time*1000 + NowLocalTime.millitm -
14|                   (long)((float)NormalVideoChunkId/(float)VideoTimeScale*1000);
15|   }
16|   StartLocalTime.time = (tempTime - tempTime%1000)/1000;
17|   StartLocalTime.millitm = tempTime%1000;
18|}
```

The server looks at its stopwatch on line 1 and checks whether the streaming process was paused or not on line 2. If the process was not paused, then of course the start time does not need to be changed. If the process was paused, the server will first set *bPause* to false to prevent executing this segment of code again in the next round, and then start to adjust the start time. On lines 4 to 5, the server reads the timestamps of the video and audio frames that it will acquire from the media file the next. On line 8, the server then compares the video timestamp with the one recorded when pausing (i.e. *LastPauseVideoFrameTimestamp*) to see if the presentation was sought. In both cases, *tempTime* is used to store the new start time in milliseconds. On lines 9 to 10, *tempTime* is equal to the current time minus the value of the sought presentation time, which is equal to *NowPauseVideoFrameTimestamp* expressed in milliseconds. On lines 13 to 14, *tempTime* is equal to the current time minus the difference between the old current time and the original start time just before pausing, which is equal to *NormalVideoChunkId* expressed in milliseconds. Lastly, on lines 16 to 17, *tempTime* is converted from milliseconds to equivalent seconds and milliseconds, which then are stored in StartLocalTime.time and StartLocalTime.millitm, respectively.

4.6.4 Bandwidth Smoothing

With the packet-based MVBA smoothing algorithm, it is possible to generate a smoother transmission plan. In the plan, how many RTP packets should be transmitted by the server at each time slot are indicated. For example, at the beginning of the transmission (first time slot), seven RTP packets should be sent out according to the transmission plan. The next time slot, say 33 ms later for the presentation with a frame rate of 29.97 frames/second, eight RTP packets should be transmitted. In other words, the number of RTP packets that should be transmitted in each slot is specified in the transmission plan. How this plan can be used to control the transmission behavior of the data streamer block of the server system is the subject of this section.

According to [21], the token bucket algorithm is used to control the amount of data that is inserted into the network. It is a transmission control mechanism that determines when the traffic can be transmitted, based on the presence of tokens in the bucket. The token bucket contains tokens, each of which represents a unit of bytes. Imaging that if the token bucket leaks a variable number of tokens periodically, and it is up to the system designer to control the period length and the number of tokens released in each period, then it is possible to let the bucket leak tokens according to the predefined plan. This concept is exactly what the proposed system uses to control the transmission behavior of the data streamer. Instead of letting the data streamer completely control when RTP packets should be injected to the network, the bandwidth smoother will tell the data streamer when to do so by releasing a proper number of tokens in each period of time according to the transmission plan generated by the packet-based algorithm. Specifically, the bandwidth smoother and the data streamer share a common semaphore named "*sem_nctu_smoothing_LeakyBucketToken*", whose count's initial value is zero. When the streaming process starts, the data streamer will obtain an RTP packet from the buffer where the packetizer places packetized RTP packets. Before the data streamer inserts this packet into the network, it will call:

```
III_SemWait(…,&StreamerInfois->sem_nctu_smoothing_LeakyBucketToken);
```

This function call causes the data streamer to wait if no token (i.e. the count of the semaphore is zero) has been released by the bandwidth smoother; otherwise, the data streamer will inject the packet into the network and the count of the semaphore will be decremented by one. On the other hand, the bandwidth smoother will examine the transmission plan generated by the packet-based MVBA algorithm to find out how many tokens the token bucket should leak in each period. For example, when a period starts and the bandwidth smoother finds that it should release seven tokens (i.e. so seven RTP packets can be inserted into the network by the data streamer), then it will call the following instruction seven times to increment the semaphore by seven.

```
III_SemPost(…,&StreamerInfois->sem_nctu_smoothing_LeakyBucketToken);
```

As a result, the data streamer will see that the count of the semaphore has grown by seven, and realize that it can transmit seven more packets.

In summary, the token bucket algorithm allows the system designer to limit or control the data transmission in a simple manner. In this work, the traffic smoothing concept is realized on the proposed system by asking both the data streamer and the bandwidth smoother (two independent but related threads) to utilize a shared semaphore. The bandwidth smoother, who acts as a traffic patrol, will increment the semaphore's count by one each time when it thinks that it is time to transmit another RTP packet. The data streamer, on the other hand, will check the count of the semaphore. If it finds that the count is zero, it will be held until the count becomes positive. If it finds that the count is equal to a certain number greater than zero, then it will inject this number of RTP packets into the network and set the count of the semaphore to zero.

4.7 Assembling Pieces

The proposed client and server systems are extremely complicated because they involve several protocols, require multiple concurrent threads, and have many modules linking together to accomplish the goals mentioned in Section 1.3. Thus, to make the aforementioned concepts of designs more understandable, this section is written to provide a summary about what has been said in this chapter, and more importantly, to describe how the pieces can be gathered to accomplish the proposed goals.

Server's Expected capabilities

Starting with reviewing what the systems can do, the server is expected to have a GUI that can interact with the program user. The GUI is responsible for collecting the user events such as button-pressing or text-typing, and then transferring these actions into the system by calling corresponding event handling functions. The server has to know how to parse MPEG-4 files in order to transport the data in a timely fashion. Moreover, the server must be able to acquire audiovisual contents from various types of cameras such as wireless or USB cameras. If these acquired contents are not in MPEG-4 format, it has to be able to transcode them into MPEG-4. In the proposed system, the transcoding is carried out with the uses of MJPEG decoder and MPEG-4 encoder. Before starting a streaming session, the server has to be able to listen and accept incoming connections. Upon establishing the connection with the client, they can start to exchange RTSP messages. Then the server can find out what media files to send and how to send them. Regarding to RTSP, the server has to manage a state machine for each of its connected clients. Upon receiving an RTSP request from one of the clients, the server has to find out the corresponding state machine, perform the state change, react to it if necessary, and reply to the client with a response indicating the result of this request. If the streaming process is started, the server has to packetize the desired media data into fixed-size RTP packets, and transmit them via UDP or TCP with the rate controlled by the traffic smoothing function that computes the transmission plan according to packet-based MVBA.

Server's Architecture

As can be observed, the server has to handle so many tasks at the same time. Thus, it is not hard to think of that several modules are needed to accomplish these tasks. Essentially, the server uses five modules: 3GP Parser, 3GP Creator, Sub Server, Server Dialog, and SP_encore.lib modules. The first two and the last modules mainly deal with MPEG-4 related tasks, such as parsing, creating virtual files for live-captured media files being accessible by the clients, and encoding raw media data into MPEG-4 format. Sub Server module is the most important one. It provides RTP and RTSP services to the system. Server Dialog module is the GUI, which interacts with the user.

Client's Expected capabilities

Having talked about the server's capabilities and architecture, it is time to turn the focus to the client. The client, compared with the server, has fewer capabilities. It has to have a GUI just like that of the server in order to interact with the user and react to the user inputs by calling corresponding event handlers. In addition, it must know how to connect to the server to establish the streaming session. After the streaming session is established, it has to prepare and mange an RTSP state machine, just like the server, and be ready to exchange RTSP messages with the server. After exchanging connection information about RTP transmission with the server via RTSP means, the client must get prepared to receive RTP packets either from UDP or TCP channel. Obviously, the client has to know how to de-packetize the received RTP packets, and parse and decode the reconstructed MPEG-4 frames in order to render the desired audiovisual contents.

Client's Architecture

Again, several modules are required to perform these jobs. There are three modules present: Sub Client, Client Dialog, and SP_decore.lib modules. The first module takes care of the RTP and RTSP related tasks. In addition, Client Dialog module is the GUI. Finally, the last module, as implied by its name, is for MPEG-4 decoding.

Operations of Threads on Server Side

After reviewing the capabilities and architectures of the client and server systems, it is time to describe how the threads employed by both systems operate to achieve the final goals. Referring to Figure 27 again, on the server side, there is a GUI thread that awaits the user inputs. When the user presses the "Start" button, several other threads will begin. First, "*Transcoder()*" will start to acquire audiovisual contents from the connected cameras, transcode or encode them into MPEG-4 format, and then place the resulted bitstreams onto the buffer where the data packetizer can access when needed. As a side note, these live-captured media data can be rendered to the GUI at the same time if desired. In addition to "*Transcoder()*" thread, "*OverallListenerThread()*" in effect will start listening to new incoming connections at port 554 via TCP connection. The listening is done by the listening socket, and whenever a connection is accepted and setup, a connected socket will be created to handle this connection. The listening socket will still exist to listen to new incoming connections. The created connected socket will be handed over to another thread called "*RTSPProcessorThread*()", which is responsible for exchanging RTSP messages with the client and reacting to them when they are valid for the current RTSP state. RTP connection information is obtained by the server when it receives SETUP request. After this information arrives and RTP channels are setup, "*StreamerletIs()*" thread will start functioning. Before doing anything, it will check the type of the target media file, if the file is live-captured, it will get the data from the buffer where "*Transcoder()*" thread places MPEG-4 bitstreams. On the other hand, if the file is stored media, it will get the data from the storage device. Whether the file is live-captured or obtained from the storage, it will be packetized into RTP packets by "*StreamerletIs()*" if the transmission is via UDP. The rate that "*StreamerletIs()*" packetizes is controlled by the token bucket realized by "*BandwidthSmoother*()" thread. "*StreamerletIs()*" will place these packets onto the buffer where "*PacketScheduler()*" can access. "*PacketScheduler()*" is a thread that inserts packets on the buffer into UDP, which in turn injects them into the network. On the other hand, if TCP is selected as the means of transportation, "*StreamerletIs()*" will not packetize media frames into RTP packet format to make the uses of timestamping and sequence-numbering provided by RTP, since TCP guarantees correct transmission (ordered arrivals and re-transmission if loss occurs). "*StreamerletIs()*" will hand over these frames to "*TunnelMain()*" thread, which inserts frames on the buffer into TCP. TCP in turn will inject them into the network.

When and by Whom Threads are Created on Server Side

Having talked about how these threads work together, it is time to see when and by whom they are created. The GUI thread is the primary thread of the process, thus it is started when the program begins. When the user presses the "Start" button, the GUI thread create both "*Transcoder()*" and "*StatusTimeFunc()*" to transcode the acquired frames and display clients' statuses, respectively. In addition, recall that an object of "*RTSPStreamServer*" class is created when "*StartServer()*" in the streaming service interface block is called by the GUI thread to start the stream server, the construction function of this class will create "*OverallListenerThread()*" to listen to and accept new incoming connections from the clients and "*PacketScheduler()*" to hand over RTP packets to UDP. Whenever "*OverallListenerThread()*" accepts and establishes a TCP connection with a new client, it will create a brand-new "*RTSPProcessorThread()*" and pass the control of this connected connection to "*RTSPProcessorThread()*", which is in charge of exchanging RTSP messages with this client and performing RTSP state changes when necessary. When "*RTSPProcessorThread()*" receives a SETUP request while in Init state, it will create "*StreamerletIs()*" to packetize media frames , and "*TunnelMain()*" if the client demands that the RTP packets are to be transmitted over TCP instead of UDP.

<u>Operations of Threads on Client Side</u>

Like the server, the client also has several threads running together to accomplish the final goals. Referring to Figure 29 again, on the top-right, the GUI thread waits for the user input. When the user presses the "Open" button, the GUI thread will try to connect to server's port 554 (handled by "*OverallListenerThread()*" on the server side) to establish a TCP connection with the server for RTSP message exchanges. After the connection is setup and ready for data transmission, "*RTSPProcessorThread()*" will take over the control of this connected socket and start to handle RTSP message exchanges with the server and perform necessary RTSP state changes. At the same time, "*VideoStatusTimeFunc()*" and "*AudioStatusTimeFunc()*" will continuously collect and display the receiving statuses for video and audio on the GUI, respectively, and also decode and render the received media frames on the GUI. After PLAY request is granted by the server and RTSP state is transited to PLAY state, "*TunnelMain()*" thread will be present to receive RTP packets if transmitted over TCP, or "*RecvThreadIsK()*" thread will be there to receive RTP packets if transmitted over UDP. If RTP packets indeed are sent via UDP channel, "*DePktThreadIsK()*" will be responsible for reconstructing the original un-packetized media frames from the received RTP packets. Depending on the data transmission channel selection, either "*TunnelMain()*" or "*DePktThreadIsK()*" will place the received frame data on the buffer where "*VideoStatusTimeFunc()*" and "*AudioStatusTimeFunc()*" can access.

When and by Whom Threads are Created on Client Side

After the explanations of how these threads work together are given, it is time to see when and by whom they are created. The GUI thread is the primary thread of the process, thus it is started when the program begins. When the user presses the “Start” button, the GUI thread will connect to port 554 of the server to establish the TCP connection for the RTSP control messages. If this connection is setup successfully, “*RTSPProcessorThread()*” will be created by the GUI to take over this connected connection and start to handle all RTSP related tasks. In addition, at this time, “*VideoStatusTimeFunc()*” and “*AudioStatusTimeFunc()*” will also be created by the GUI. If the user desires that the media data are to be delivered via TCP, “*TunnelMain()*” thread will be created by the GUI to receive them from TCP when SETUP request is issued and sent to the server. Moreover, “*RecvThreadIsK()*” and “*DePktThreadIsK()*” will also be created by the GUI to be able to receive RTP packets via UDP and de-packetize those received RTP packets.

Summary

Section 4.7 can be summarized as follows. Referring to Figure 34, when both the client and the server systems first start up, only GUI threads will be up and running since they are the primary threads of the processes. These GUI threads will sit there waiting for the user inputs, such as button pressing or text editing. Referring to Figure 35, when the “Start” button on the server side is clicked to start the streaming service, four more threads will be created. Their functions are described earlier. In particular, “*OverallListenerThread()*”, which is designed to establish RTSP connections with multiple clients, will have a TCP listening port listening and accepting any incoming connection request arriving at port 554. The GUI thread of the client upon being requested by its user, will try to establish a TCP connection with “*OverallListenerThread()*”. Referring to Figure 36, after this TCP connection is established, the control of the connected socket on the server side will be passed from “*OverallListenerThread()*” to “*RTSPProcessThread()*”. Similarly, on the client side, the control of the connected socket will be passed from GUI thread to “*RTSPProcessThread()*”. Therefore, “*RTSPProcessThread()*” threads on both sides can start their RTSP message exchanges via TCP channel.

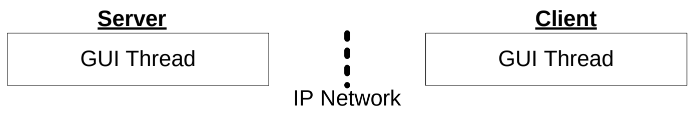


Figure 34 – Threads present when both server and client programs start

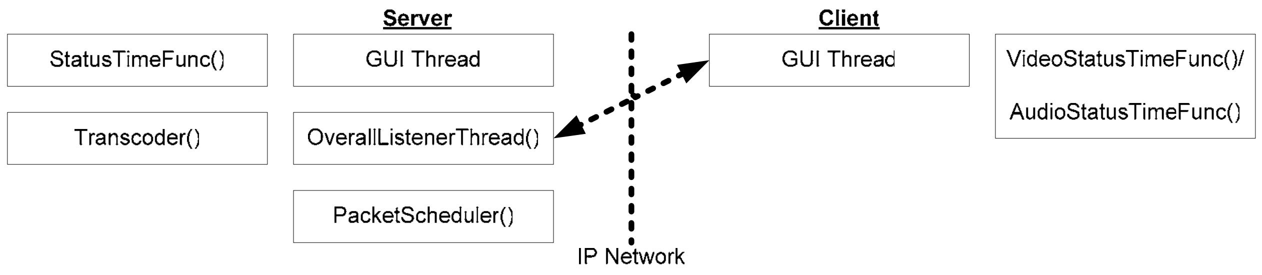


Figure 35 – Threads present when server starts its service and client connects to it

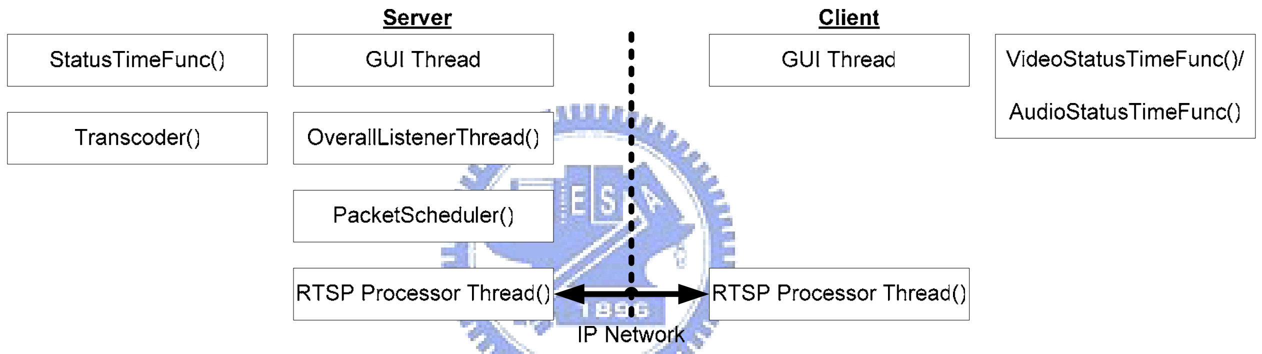


Figure 36 – Threads present when RTSP message exchanges are in progress

When SETUP messages are being exchanged by the client and the server, the RTP channel used to transport the RTP packets should be setup. If UDP is selected as the RTP channel, referring to Figure 37, on the client side, “RecvThreadIsK()” and “DePktThreadIsK()” will be created to receive RTP packets and de-packetize received RTP packets, respectively. On the server side, if live-captured data are to be transmitted, then “StreamerletIs()” will acquire media frames from “Transcoder()” thread, packetize them into RTP packets, and hand over these packets to “PacketScheduler()”, which in turn will transmit these packets to “RecvThreadIsK()” on the client side. If stored media are to be streamed, then “StreamerletIs()” will acquire media frames from the storage device, packetize them into RTP packets with the rate regulated by “BandwidthSmoother(), and hand over these packets to “PacketScheduler()”, which in turn will transmit these packets to "RecvThreadIsK()" on the client side. On the other hand, if TCP is preferred, referring to Figure 38, then on both sides, "TunnelMain()" threads will be created. The TCP connection between them is setup in the way similar to the RTSP channel. On the server side, if live-captured data are to be transmitted, then "StreamerletIs()" will acquire media frames from "Transcoder()" thread. On the other hand, if stored media are to be streamed, then "StreamerletIs()" will acquire media frames from the storage device. In either case, "StreamerletIs()" will hand over these frames to "TunnelMain()", which in turn will transmit these packets to "TunnelMain()" on the client side.

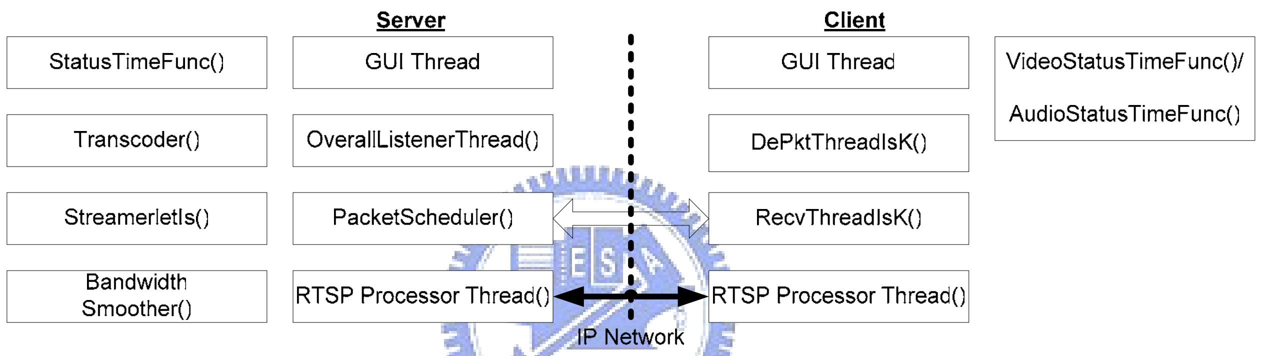


Figure 37 – Threads present when both RTSP and RTP via UDP are in progress

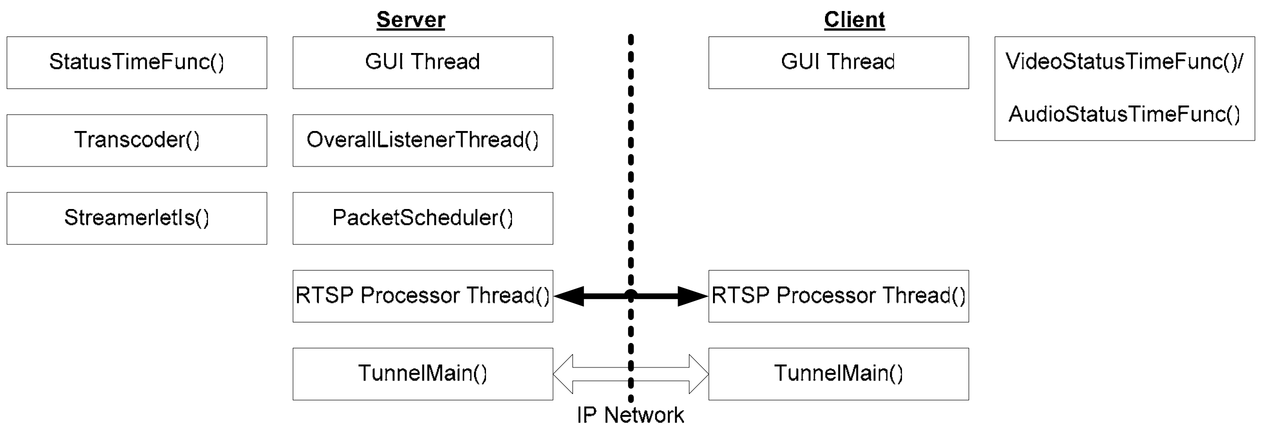


Figure 38 – Threads present when both RTSP and RTP via TCP are in progress

5 Usages of Modules

This chapter gives descriptions about how to use the implemented modules to develop the server and the client systems. In particular, the usages of 3GP Parser and Creator modules, Sub Server module, and Sub Client module will be described in detail.

5.1 Using 3GP Parser module

This section gives information about how to use 3GP Parser module. Some simple examples will be given to illustrate how this module can be used. After that, a complete description of its API will be given.

5.1.1 Getting started with 3GP Parser module

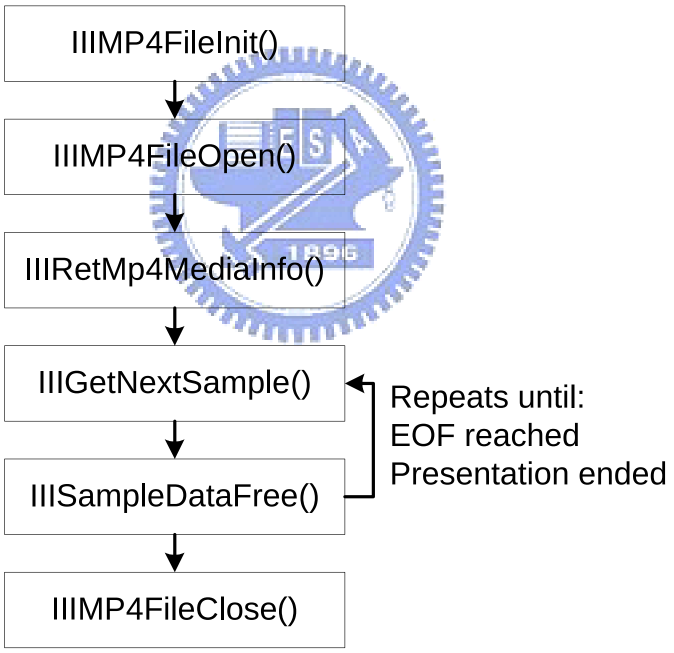


Figure 39 – Typical usage of 3GP Parser module

Referring to Figure 39, to deal with a MPEG-4 file, one has to define a pointer for Mp4FileInfo structure, and specify the name of the file to be opened, its data type (MP4 or THREEGPP), and the memory mode (III_MEM_NROMAL_MODE, III_MEM_SAVE_MODE, III_MEM_ULTRA_SAVE_MODE). For example, the code can be written as:

```
Mp4FileInfo   *mp4_file_info;
char buffer[] = “starwar.mp4”;
FILEFORMAT FILETYPE = MP4;
```

Next, the desired MPEG-4 file can be initialized and opened as the following:

```
IIIMP4FileInit(&mp4_file_info, buffer, III_MEM_SAVE_MODE);
IIIMP4FileOpen(&mp4_file_info,FILETYPE);
```

To retrieve the media information about the file (for example, the timescales of the video/audio), one can write the code as follows:

```
IIIRetMediaInfo *mp4vinfo;
IIIRetMediaInfo *mp4ainfo;
IIIRetMp4MediaInfo(mp4_file_info,
                   mp4_file_info->pCurrentVideoTrackis->TrackID,
                   VideoTrackCodeName,
                   &mp4vinfo);
IIIRetMp4MediaInfo(mp4_file_info,
                   mp4_file_info->pCurrentAudioTrackis->TrackID,
                   AudioTrackCodeName,
                   &mp4ainfo);
VideoTimeScale = mp4vinfo->TimeScale;
AudioTimeScale = mp4ainfo->TimeScale;
```

It is obvious to realize that the media information is placed in the memory location pointed by mp4vinfo or mp4ainfo.

After acquiring the desired media information, remember to release the resources associated with mp4vinfo and mp4ainfo.

```
IIIMp4MediaInfoFree(&mp4vinfo);
IIIMp4MediaInfoFree(&mp4ainfo);
```

To get video/audio frames, one can perform the following actions:

```
pRetSampleData video_buffer;
pRetSampleData audio_buffer;
IIIGetNextSample(mp4_file_info,
                  mp4_file_info->pCurrentVideoTrackis->TrackID,
                  VideoTrackCodeName,
                  &video_buffer);
IIIGetNextSample(mp4_file_info,
                  mp4_file_info->pCurrentAudioTrackis->TrackID,
                  AudioTrackCodeName,
                  &audio_buffer);
```

The acquired frame will be placed in video_buffer or audio_buffer.

After using this frame, the resources associated with it can be released as follows:

```
IIISampleDataFree(&video_buffer);
IIISampleDataFree(&audio_buffer);
```

Finally, after finishing using the media file, one can close it by calling:

```
IIIMP4FileClose(&mp4_file_info,FILETYPE);
```

5.1.2 The complete API

Here, the complete API of 3GP Parser module will be described.

- *int IIIMP4FileInit(Mp4FileInfo** FileInfo, char* Filename, int nMemMode)*
  - It initializes the MPEG-4 file named *Filename* by opening the file, checking out the file's validity, finding out the size of the file, and setting the memory consuming mode.
- *int IIIMP4FileOpen(Mp4FileInfo** FileInfo, FILEFORMAT filetype)*
  - It opens the selected MPEG4 file that is of *filetype* type by checking if the atom tree structure fits the correct requirements or not, allocating memory space for the non-container atoms, getting atoms from the file, filling the table entries within non-fix-length atoms, taking all track atoms, and classifying them and storing them into video track handler, audio track handler, and other handler.

- *int IIIMP4FileClose(Mp4FileInfo** lpFileInfo, FILEFORMAT filetype)*
  - It releases the resources associated with *lpFileInfo*.
- *int IIIRetMp4MediaInfo(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType, pIIIRetMediaInfo *pptmpIIIRetMediaInfo)*
  - It acquires the media information about the *nTrackID* in *lpFileInfo* that is of *nTrackType* type. The information acquired includes the timescale, the duration, the atom length, the average bitrate, the sampling rate, and so on.
- *int IIIMp4MediaInfoFree(pIIIRetMediaInfo *pptmpIIIRetMediaInfo)*
  - It releases the resources associated with *pptmpIIIRetMediaInfo*.
- *int IIIGetNextSample(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType, pRetSampleData *pptmpRetSampleDatais)*
  - It gets the next video/audio sample from the *nTrackID* in *lpFileInfo* that is of *nTrackType* type.
- *int IIIGetSample(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType, int nRequesdTime, pRetSampleData *pptmpRetSampleDatais)*
  - It gets the video/audio sample that has the timestamp equal to *nRequesdTime* from the *nTrackID* in *lpFileInfo* that is of *nTrackType* type.
- int IIIGetNextSampleTimeStamp(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType, int *pnRetTimestampis)
  - It gets the timestamp of the next sample from the *nTrackID* in *lpFileInfo* that is of *nTrackType* type.
- *int IIISetMemConsumeMode(Mp4FileInfo* pFileInfo,int nModeis)*
  - It sets the memory consuming mode. *nModeis* can be one of the following:
    - ◆ *III_MEM_NROMAL_MODE*
    - ◆ *III_MEM_SAVE_MODE*
    - ◆ *III_MEM_ULTRA_SAVE_MODE*
- int IIINextKeyFrame(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType, pRetSampleData *pptmpRetSampleDatais)
  - It returns the frame data of the very next key frame from the *nTrackID* in *lpFileInfo* that is of *nTrackType* type.
- int IIIPrevKeyFrame(Mp4FileInfo* lpFileInfo, unsigned int nTrackID, int nTrackType,

pRetSampleData *pptmpRetSampleDatais)

- It returns the frame data of the previous key frame from the *nTrackID* in *lpFileInfo* that is of *nTrackType* type.

- int IIISampleDataFree(pRetSampleData *pptmpRetSampleDatais)
  - It releases the resources associated with the sampled data *pptmpRetSampleDatais*.

5.2 Using 3GP Creator module

3GP Creator module is needed to create a new MPEG-4 media file. This section gives information about how to use 3GP Creator module. Some simple examples will be given to illustrate how this library can be used.

To create a new MPEG-4 video file from the bitstream located at position *TargetBuffer*, the following should be done:

```
MP4_local_file*    m_SaveMP4File;
FileName[]= “starwar.mp4”;
int width;
int height;
int nFormatLength;
MP4FileNew(&m_SaveMP4File, FileName);
GetVideoInfo(TargetBuffer,&width,&height,&nFormatLength);
CreateTrack(m_SaveMP4File, VideoTrackCodeName, width, height,
                FRAMERATE, TIMESCALE, 1, 0, 0, 0, nFormatLength,
                TargetBuffer, "temp.m4v", 1);
```

To add a new video frame to the same MPEG-4 media file, the following function should be called:

```
AddVideoFrame(m_SaveMP4File,TargetLength,TargetBuffer, TimeStamp,
                     IsKeyFrame);
```

Note that *TimeStamp* shows the timestamp of the frame and *IsKeyFrame* notifies that whether this frame is an I-Frame.

Finally, after all the frames are added to the newly created MPEG-4 file, it can be closed by calling:

```
MP4FileFlush(m_SaveMP4File);
```

5.3 Using Sub Server module

Since the streaming server has to stream media data to multiple clients in accordance with RTP and exchange control messages with them in compliance with RTSP, it needs help from a special module other than Windows® standard library winsock.h and winsock2.h, which only provide fundamental TCP/UDP services. Sub Server module is such a great module. By using the following interface provided by this module, the server program can start or stop its streaming action and communicate with its clients with ease.

5.3.1 Getting started with Sub Server module

To create and start a simple streaming server which can transmit stored media files to single or multiple clients can follow the code listed below:

```
void *test_stream_server = NULL;
RTSPSERVERINFO test_server_info;
test_server_info.lpRootDir = (char *)malloc(256 * sizeof(char));
sprintf(test_server_info.lpRootDir,"C:\\video");
test_server_info.Port = 554;
test_server_info.VideoPort = 2030;
test_server_info.AudioPort = 2020;
test_server_info.source_from_capture = 1;
test_server_info.ServerStatus = SERVER_RUNNING;
test_server_info.MaxClient = 150;
StartServer(&test_stream_server, &test_server_info);
```

To stop running the streaming server, issue the following line:

```
StopServer(&test_stream_server);
```

To get the number of connecting clients and their corresponding status, run the following code:

```
int client_number;
CLIENT_STATUS client_status;
```

```
GetClientNumber(&test_stream_server, &client_number);
for( int i=1;i<=client_number;i++){
      GetClientStatus(&test_stream_server, i, &client_status);
      printf("%s: ", client_status.peerAddr);
      switch(client_status.rtsp_state){
            case 0:
                  printf("INIT\n");
                  break;
            case 1:
                  printf("PAUSE\n ");
                  break;
            case 2:
                  printf("PLAY(%dk) \n ", client_status.transmitRate);
                  break;
            default:
                  break;
      }
 }
```

5.3.2 The complete API

- StartServer(void ** StreamServer, LPRTSPSERVERINFO in_server_info)
  - It starts a streaming server which is characterized by *in_server_info* that specifies the root dir, the RTSP port, video/audio ports, camera IP addresses, the FGS transfer rate, the maximum number of clients, and so on.
- StopServer(void ** StreamServer)
  - It stops the running streaming server.
- CaptureToServer(void ** StreamServer,int stream_type, int input_channel, int real_time_capture,MediaItem *mediabuffer)
  - It feeds the real-time captured and compressed video/audio data acquired from cameras to the output buffer, where data frames are waiting to be transmitted by other thread upon requested.

- GetClientNumber(void ** StreamServer,int * client_number)

  - It returns to the caller the number of clients connecting to the streaming server.
- GetClientStatus(void ** StreamServer,int client_number, CLIENT_STATUS * client_status)

  - It returns to the caller the status of specified client.

The interface functions listed above look simple and in fact, can be used in an effortless way. This user-friendly feature is accomplished by hiding several extremely complicated sub-blocks of functions from the GUI designer. These sub-blocks are:

- Interface
  - It provides simple interface functions to the server program so the server can stream the media data and exchange control messages with clients in an effortless way.
- Stream
  - It contains a OverallListenerThread() to handle connection setup requests from clients and a RTSPProcessorThread() for each client to handle client's control messages.
- Net
  - It is one level above TCP/IP. The actual streaming of video/audio data is scheduled and performed by this sub-block.
- RTSP
  - It contains ParseRTSP() for the server to parse RTSP control messages

  It deals with handling RTSP related messages.
- III Services
  - It provides fundamental functions to support mutex, semaphores, threads, time, memory management, and several base classes.
- Input
  - It provides two types of classes for *the prog*rammer to use. The first is *MediaBuffer* and the second is *MediaItem*.
- TCP Tunnel
  - It deals with the packets sent via TCP tunnel instead of the UDP method.

5.4 Using Sub Client module

Since the client receives media data to from the streaming server in accordance with RTP and exchange control messages with the server in compliance with RTSP, it needs help from a special module other than Windows® standard library winsock.h and winsock2.h, which only provide fundamental TCP/UDP services. Sub Client module is such a great module. By using the following interface provided by this module, the client program can receive streaming data from the server and communicate with the server with ease.

5.4.1 Getting started with Sub Client module

Getting connected to a streaming server is quite easy when using Sub Client module, because this module offers a very neat and simple interface for the system designer to work with. First of all, we want to create a client and setup the connection between the client and the server:

```
void * test_stream_client;
int ReTransFlag,TCP_tunnel_Flag;
char URL[]=”rtsp://guest:guest@192.168.0.100/starwar.mp4”;
StreamCreateClient(&test_stream_client);
StreamOpenConnection(&test_stream_client,0,0,URL,0,100);
```

Note that the second and the third parameters of StreamOpenConnection() denote whether the retransmission flag is on, and if this connection is via TCP tunnel or not, respectively.To get a video frame from received buffer, the following function has to be called:

```
MediaItem *ucBuffer;
unsigned long data_readed;
unsigned long time_stamp;
StreamGetVideoFrame(&test_stream_client, &ucBuffer, &data_readed, &time_stamp,0);
```

The following code allows you to get the current video frame timestamp, the duration of video frames in received buffer, the timescale of the video, and the current average transmission bitrate.

```
unsigned long LastTimeStamp;
unsigned long TimeScale;
double dLossRate;
int transmitRate;
StreamGetCurrentVideoTime(&test_stream_client, &temptest);
StreamGetBufferVideoTime(&test_stream_client, &time_stamp, &LastTimeStamp);
StreamGetTimeScale(&test_stream_client, VIDEOSTREAM, &TimeScale);
StreamGetLossRate(&test_stream_client, VIDEOSTREAM, &dLossRate);
StreamGetTransmitRate(&test_stream_client, VIDEOSTREAM, &transmitRate);
```

To terminate the connection between the server and the client, run the following lines:

```
StreamCloseConnection(&test_stream_client);
StreamReleaseClient(&test_stream_client);
```

5.4.2 The complete API

- StreamCreateClient(void ** StreamClient)
  - It creates a streaming client by allocating necessary media buffer and creating the *RTSPStreamClient* object.
- StreamOpenConnection(void ** StreamClient, int ReTransFlag, int TCP_tunnel_Flag, char * url, unsigned int StartSec, unsigned int EndSec)
  - It opens the connection between the server and the client.
- StreamPauseConnection(void ** StreamClient)
  - It pauses the connection between the server and the client.
- StreamSeekConnection(void ** StreamClient, int *VideoSec, int accurate)
  - It asks the server to get ready to transmit the media data whose presentation time is specified.
- StreamPlayConnection(void ** StreamClient)
  - It requests the server to start to transmit the desired media data.
- StreamGetSeekTime(void ** StreamClient, int *VideoSec)
  - Because it is possible that a user might seek the media data to an incorrect time of

presentation, this function is needed to correct the erroneous seek time and return the corrected one.

- StreamGetAudioFrame(void ** StreamClient, MediaItem **ucBuffer, unsigned long *nActualRead, u_int32 *nTimeStamp, int goback)
  - It gets the next available received audio frame.
- StreamGetVideoFrame(void ** StreamClient, MediaItem **ucBuffer, unsigned long *nActualRead, u_int32 *nTimeStamp, int goback)
  - It gets the next available received video frame.
- StreamGetCurrentAudioTime(void ** StreamClient, u_int32 *TimeStamp)
  - It returns the current audio time stamp.
- StreamGetCurrentVideoTime(void ** StreamClient, u_int32 *TimeStamp)
  - It returns the current video time stamp.
- StreamGetBufferAudioTime(void ** StreamClient, u_int32 *TotalTimeStamp, u_int32 *LastTimeStamp)
  - It returns the duration of received audio frames currently placed in buffer.
- StreamGetBufferVideoTime(void ** StreamClient, u_int32 *TotalTimeStamp, u_int32 *LastTimeStamp)
  - It returns the duration of received video frames currently placed in buffer.
- StreamGetTimeScale(void ** StreamClient, int stream_type, DWORD *TimeScale)
  - It returns the timescale of the specified media data.
- StreamGetDuration(void ** StreamClient, int stream_type, DWORD *Duration)
  - It returns the total duration of the specified media data.
- StreamGetState(void ** StreamClient,int *rtsp_state)
  - It returns the current state of the client.
- StreamGetLossRate(void ** StreamClient,int stream_type,double *dLossRate)
  - It returns the average loss rate of the streaming session.
- StreamGetTransmitRate(void ** StreamClient,int stream_type, int *transmitRate)
  - It returns the average transmission bitrate of the streaming session.
- StreamCloseConnection(void ** StreamClient)
  - It closes the connection between *StreamClient* and the server.
- StreamReleaseClient(void ** StreamClient)

- It releases the resources associated with *StreamClient*.

- StreamGetVideoEsdsData(void ** StreamClient,BYTE *DataOut, DWORD *OutLen)
  - It acquires video decoder specific information data.
- StreamGetAudioEsdsData(void ** StreamClient,BYTE *DataOut, DWORD *OutLen)
  - It acquires audio decoder specific information data.
- StreamGetFrame(void ** StreamClient,MediaItem **ucBuffer, unsigned long *nActualRead,u_int32 *nTimeStamp,int stream_type, int goback)
  - It gets the next frame of specified media data. It is used by both *StreamGetAudioFrame()* and *StreamGetVideoFrame()* which specify their *stream_type* when calling *StreamGetFrame()*.
- StreamGetLastVideoTime(void ** StreamClient,u_int32 *TimeStamp)
  - It returns the timestamp of the previously acquired video frame.
- StreamGetLastAudioTime(void ** StreamClient,u_int32 *TimeStamp)
  - It returns the timestamp of the previously acquired audio frame.
- StreamGetStreamCodecInfo(void ** StreamClient,int *VideoCodec,int *AudioCodec)
  - It returns the information of the codec including whether the data contains audio or video, and if applicable, their compression methods.

The interface functions listed above look simple and in fact, can be used in an effortless way. This user-friendly feature is accomplished by hiding several extremely complicated sub-blocks of functions from the GUI designer. These sub-blocks are:

- Interface
  - It provides simple interface functions to the server program so the server can stream the media data and exchange control messages with clients in an effortless way.
- Stream
  - It defines a class of objects named *RTSPStreamClient*, which defines all the functions needed to handle a streaming client, necessary variables, and memory buffers.
- Net
  - It is one level above TCP/IP. The actual streaming of video/audio data is

scheduled and performed by this sub-block.

- RTSP
  - It contains ParseRTSP() for the server to parse RTSP control messages

  It deals with handling RTSP related messages.
- III Services
  - It provides fundamental functions to support mutex, semaphores, threads, time, memory management, and several base classes.
- Input
  - It provides th types of classes for the programmer to use. They are *MediaInputStream*, *MediaBuffer* and *MediaItem*.
- TCP Tunnel
  - It deals with the packets sent via TCP tunnel instead of the UDP method.

6 System Execution

In this chapter, the executions of the proposed system are introduced and shown. In particular, the pictures of server interface and the client interface, and the results of executions are given and described in detail.

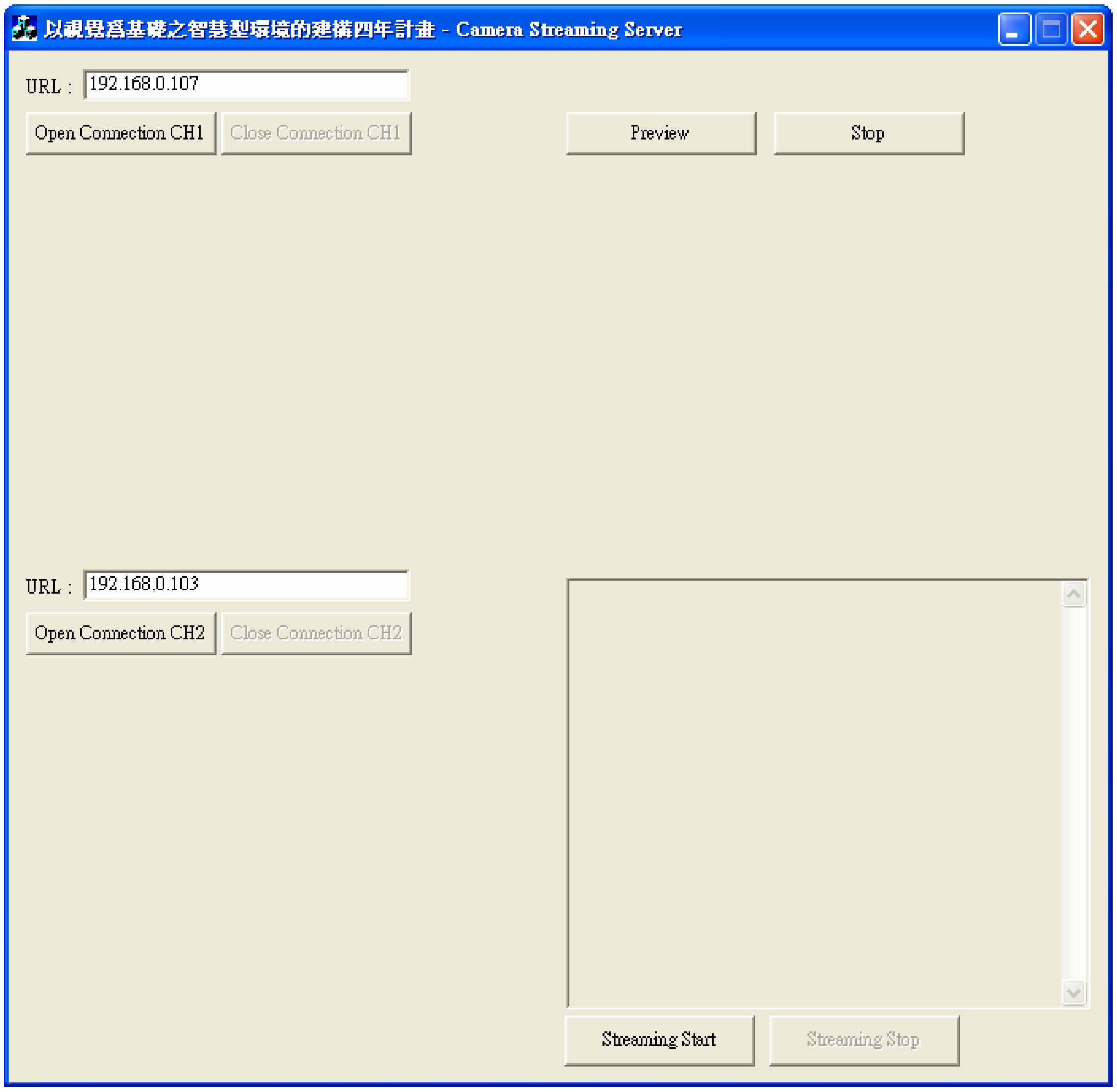


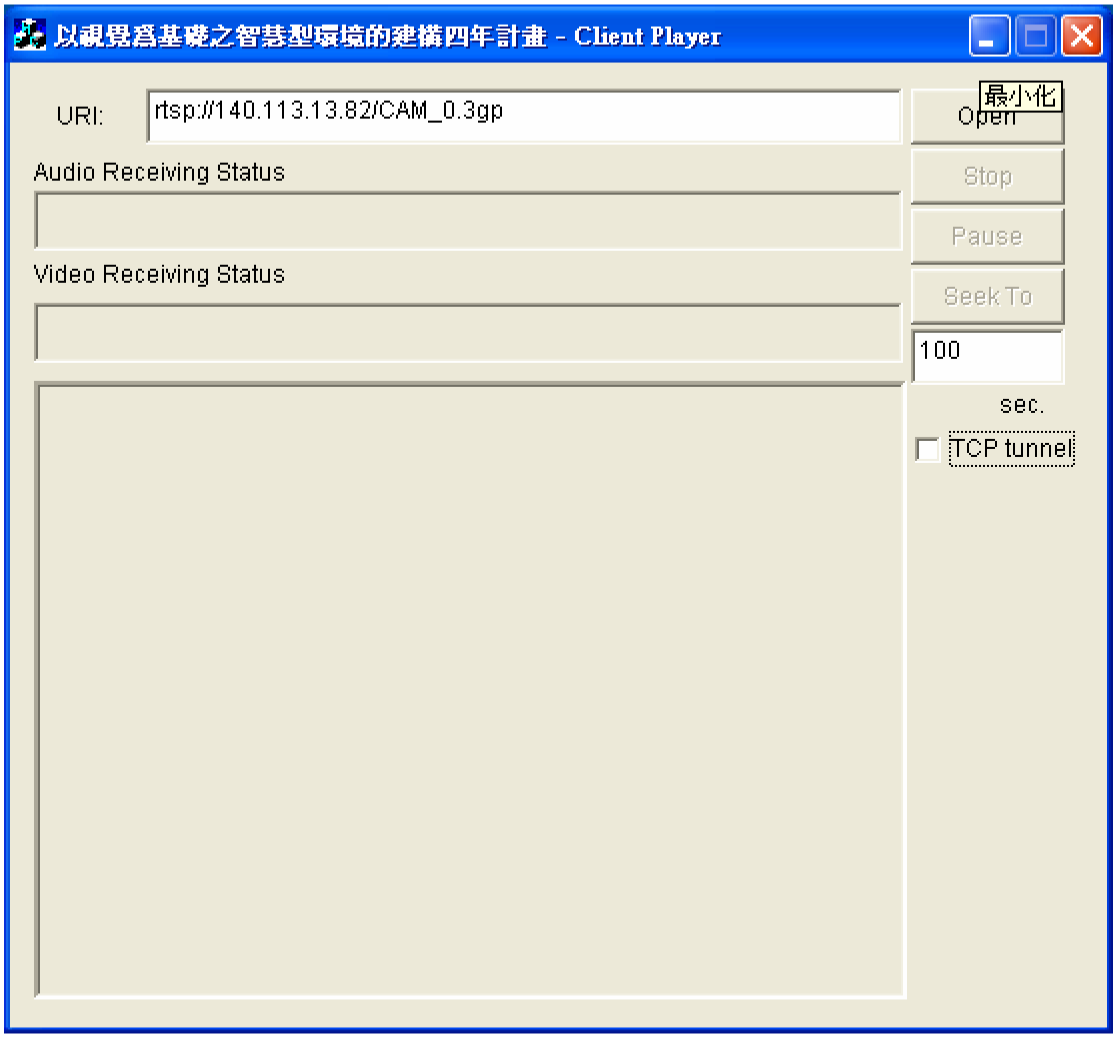
Figure 40 – Server interface

6.1 Server Interface

When the streaming server program is started, its outlook is as shown in Figure 40. On the left, there are two “URI” edit fields for the user to input the IP addresses of wireless IP cameras. Before streaming service begins, the user must click on “Open Connection CH1” and “Open Connection CH1” buttons in order to get images from these cameras. In addition, if the USB camera is also connected to the server PC and the audiovisual contents that it captures are desired, the “Preview” button also has to be clicked. After these cameras are connected, their images can be shown on the GUI for previewing purpose. To start serving the client, “Streaming Start” button should be pressed. To stop the service at any time, “Streaming Stop” button can be pressed. Finally, the gray edit field on the right side is for displaying the connection status. Its uses will become clear in Section 6.3.

6.2 Client Interface

When the streaming client program is started, its outlook is as shown in Figure 41. On the top left, “URI” edit field is for the user of the client to input the URI of the desired media file. The URI includes the IP address of the server and the name of the file. On the bottom there is a gray picture box. This picture box is where the received visual contents will be rendered. To connect to the server, “Open” button has to be clicked. To stop the connection with the server, “Stop” button is to be pressed. To pause the streaming process, “Pause” should be clicked. To resume the paused streaming process, “Pause” should be clicked again. To forward or backward the presentation, the user has to pause the streaming process, enter the desire presentation time in the edit field on the right side, and then click on “Seek” button. If the user demands that the data should be transmitted via TCP instead of UDP, then the check box marked “TCP Tunnel” should be checked. Finally, the gray edit fields marked “Audio Receiving Status” and “Video Receiving Status” are for displaying the receiving statuses. Their uses will become clear in Section 6.3.



rtsp://140.113.13.82/CAM_0.3gp

100

Figure 41 – Client interface

6.3 Results of Executions

When the server is connecting to two wireless cameras and one USB camera

Figure 42 shows the outlook of the server program when two wireless cameras and one USB camera are connected. Obviously, the images that they capture are being previewed in the picture boxes of the GUI.

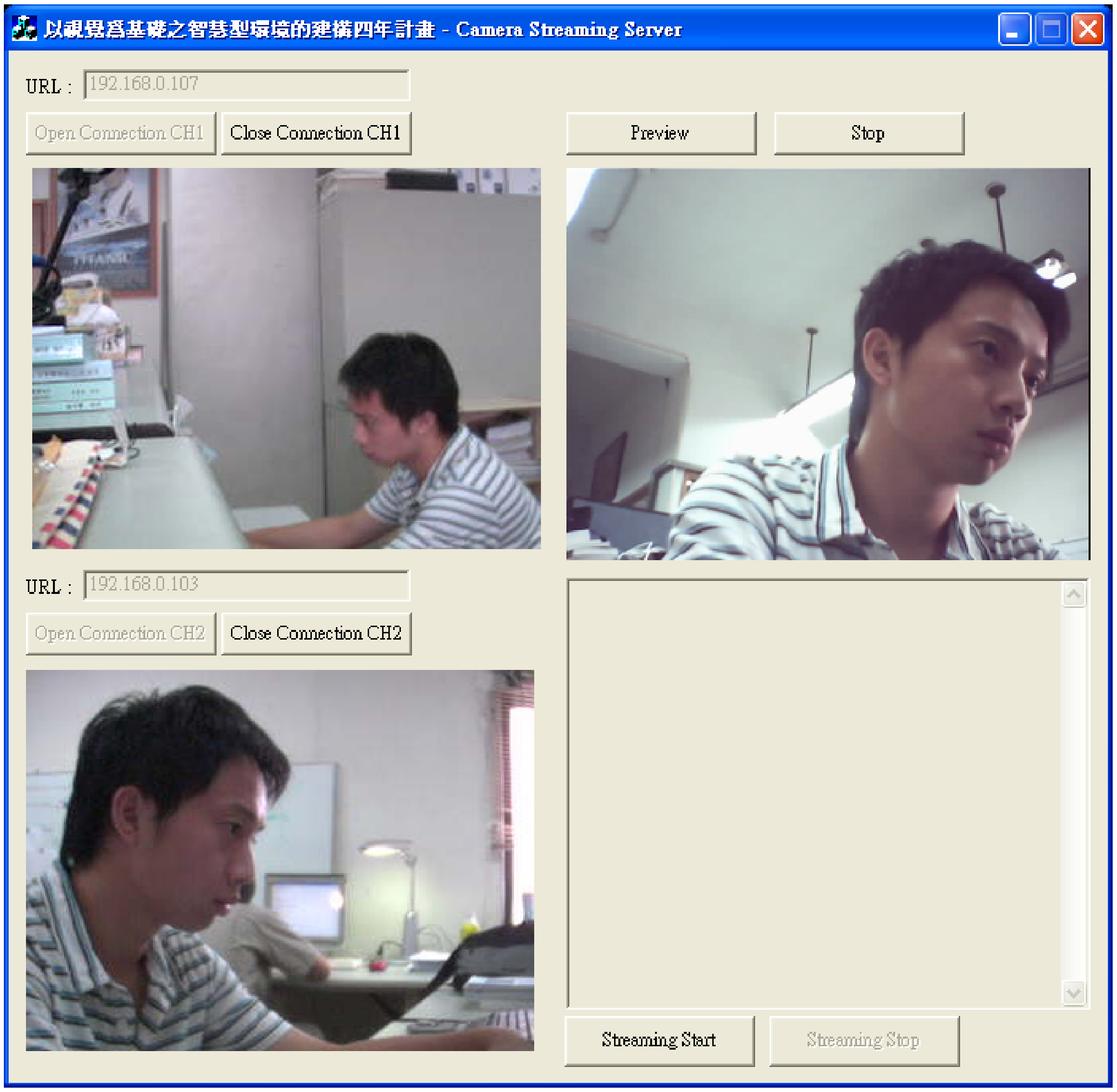
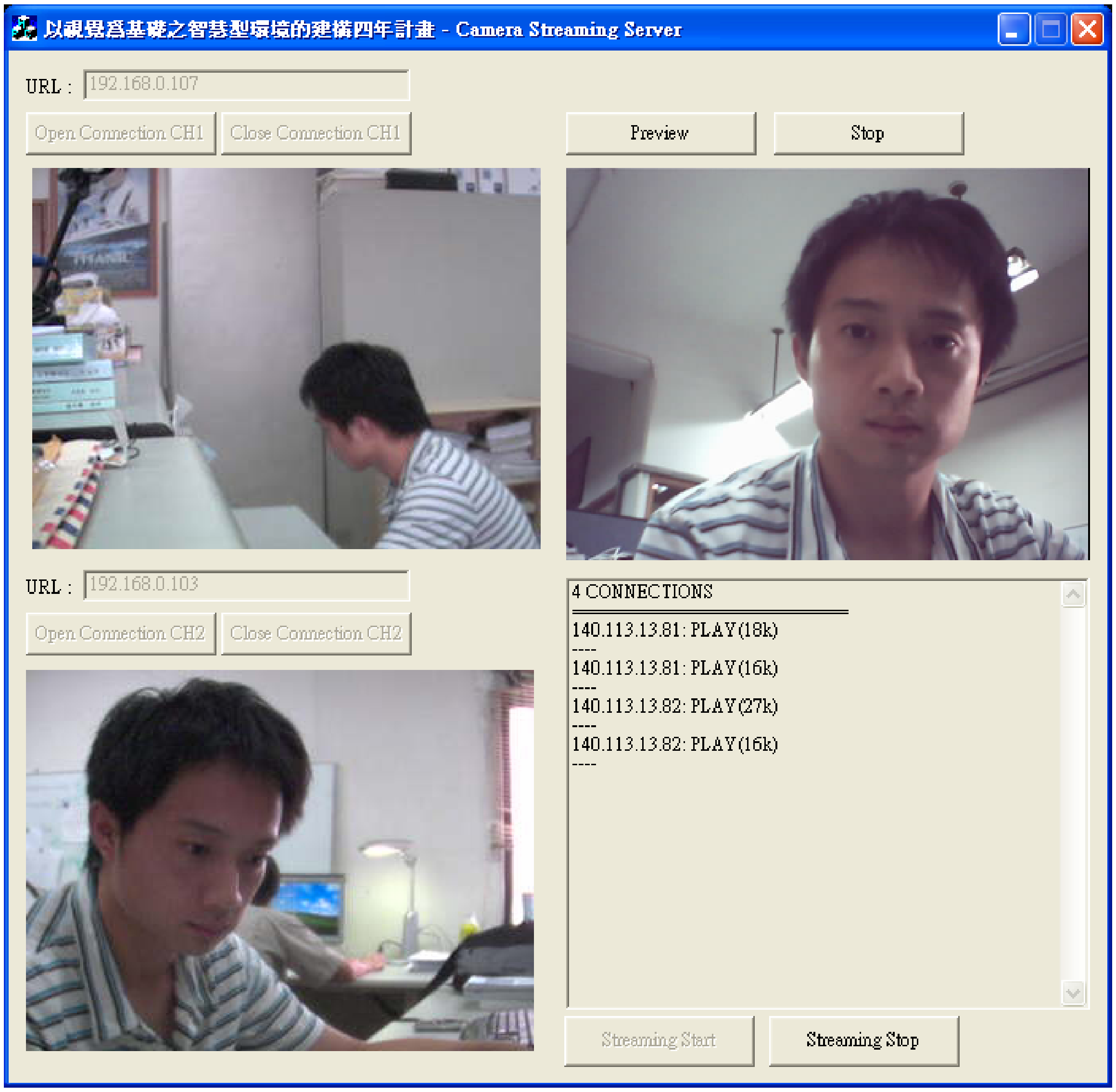


Figure 42 – Server interface when connecting to two wireless cameras and one USB camera

When four clients are connecting to the server:

In Figure 43, the streaming service is started by clicking on "Streaming Start" button. At this time, a total of four clients are connecting to the server as shown on the edit field on the bottom right. Their IP addresses, and corresponding RTSP states and transmission rates are also displayed.



URL : 192.168.0.107

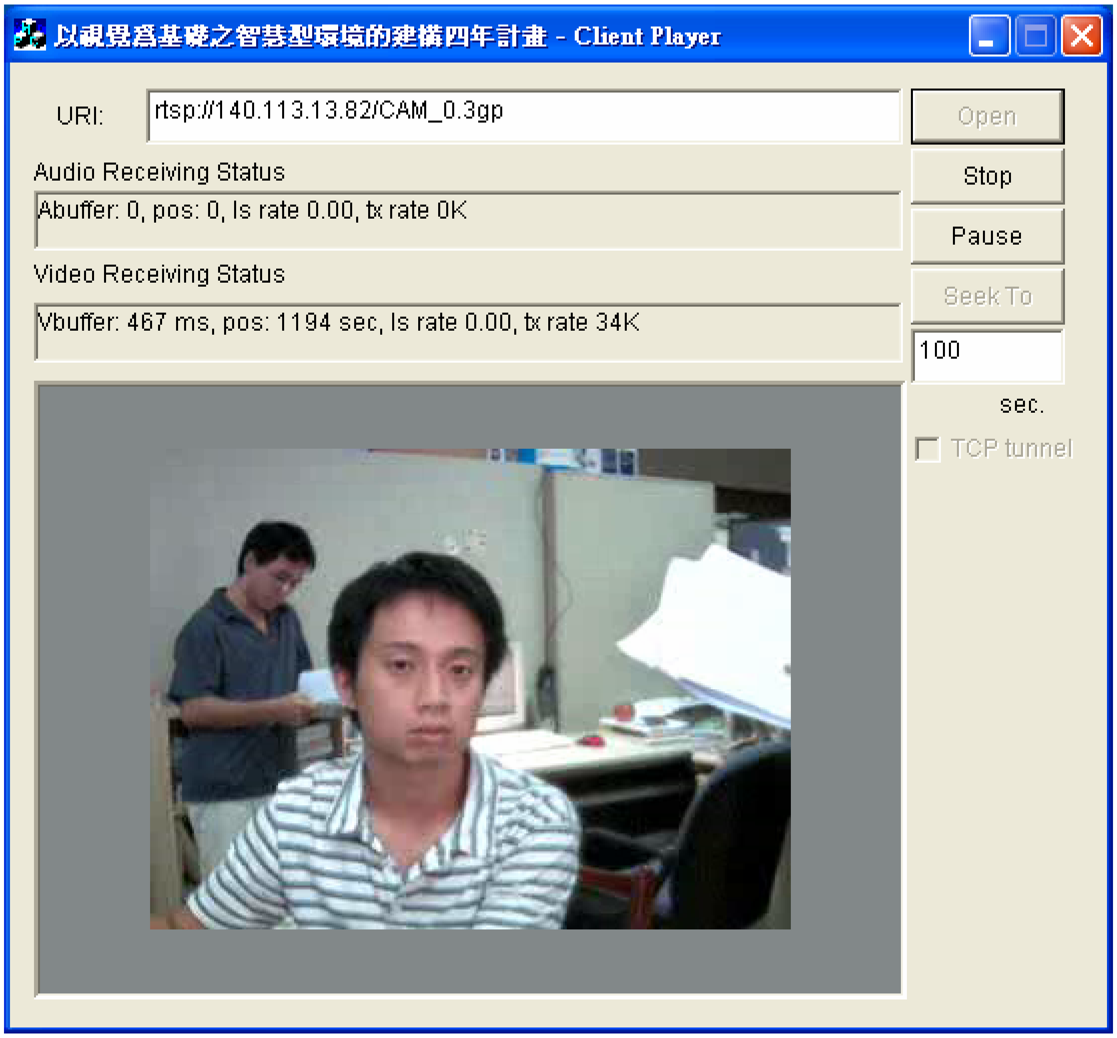
URL : 192.168.0.103

```
4 CONNECTIONS
140.113.13.81: PLAY(18k)
----
140.113.13.81: PLAY(16k)
----
140.113.13.82: PLAY(27k)
----
140.113.13.82: PLAY(16k)
----
```

Figure 43 – Server interface when four clients are connecting to it

When client is receiving live-captured contents:

In Figure 44, the client program is connecting to the server to receive live-captured contents by specifying URI of the desired media file and pressing “Open” button. As can be observed, the receiving statuses for audio and video are shown separately. These statuses include the buffer lengths in milliseconds, the current presentation times, the loss rate, and the transmission rates. As a side note, since the cameras used here do not capture sounds, the audio receiving status field does not show anything.


以視覺為基礎之智慧型環境的建構四年計畫 - Client Player

URI: rtsp://140.113.13.82/CAM_0.3gp

Audio Receiving Status

Abuffer: 0, pos: 0, ls rate 0.00, tx rate 0K

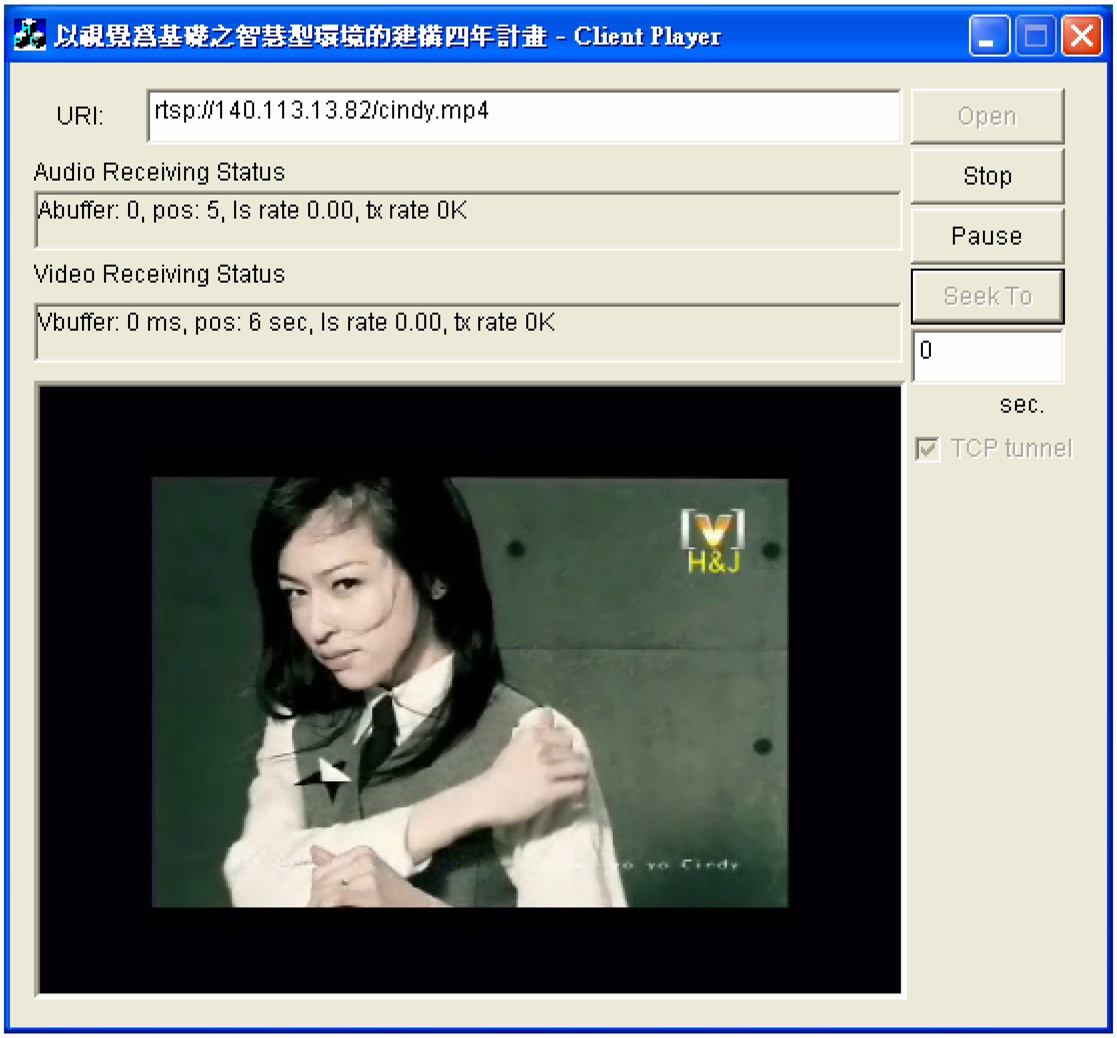
Video Receiving Status

Vbuffer: 467 ms, pos: 1194 sec, ls rate 0.00, tx rate 34K

Figure 44 – Client interface when acquiring live-captured contents

When client is receiving stored contents:

In Figure 45, the client program is connecting to the server to receive stored media contents by specifying URI of the desired media file and pressing “Open” button. Note that because “cindy.mp4” includes both video and audio, the receiving statuses for both are shown.


以視覺為基礎之智慧型環境的建構四年計畫 - Client Player

URI: rtsp://140.113.13.82/cindy.mp4

Audio Receiving Status

Abuffer: 0, pos: 5, ls rate 0.00, tx rate 0K

Video Receiving Status

Vbuffer: 0 ms, pos: 6 sec, ls rate 0.00, tx rate 0K

Figure 45 – Client interface when acquiring stored contents

When Windows® Media Player® is used as client:

In Figure 46, on the left, the Windows® Media Player® is playing the lived-captured contents, while on the right, it is playing the stored media contents.

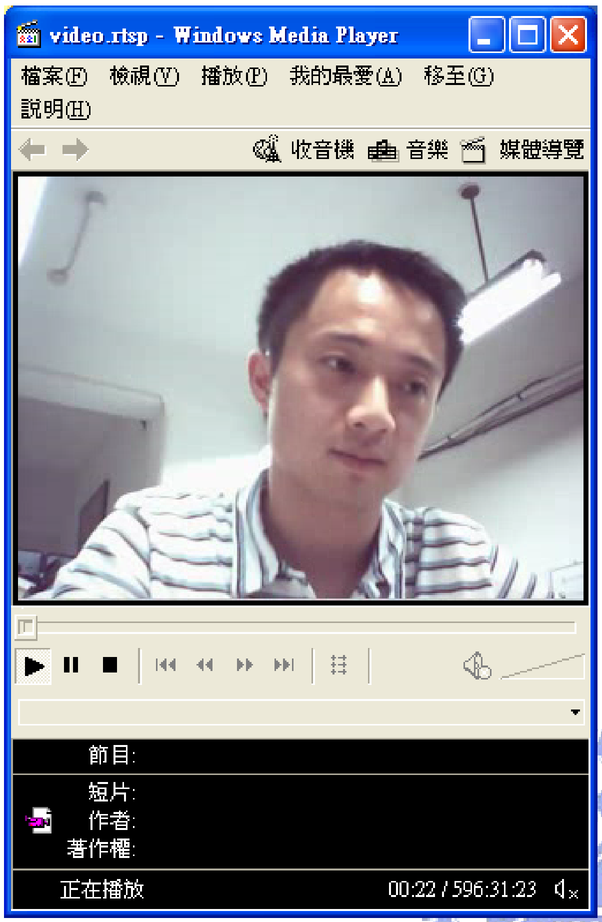
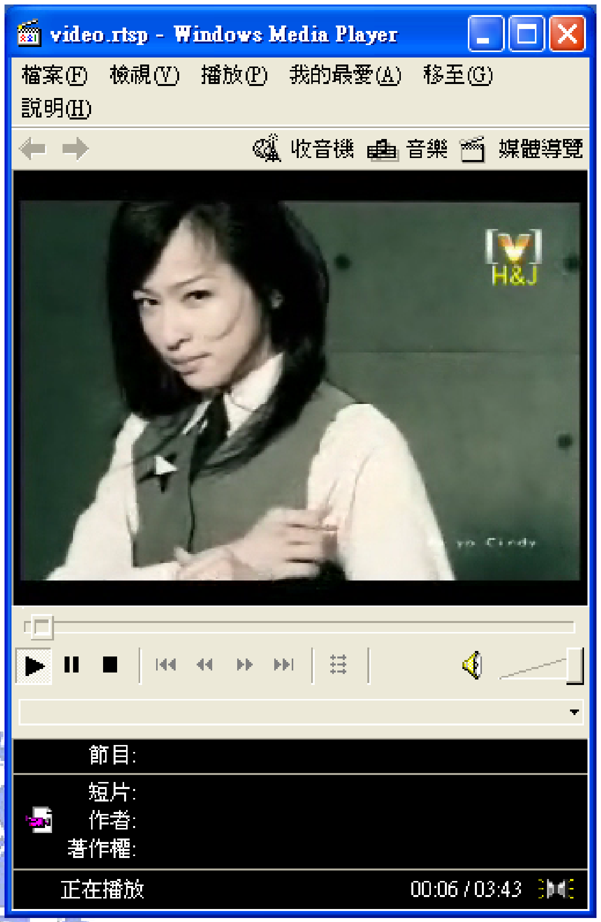

video.rtsp - Windows Media Player
檔案(F) 檢視(V) 播放(P) 我的最愛(A) 移至(G)
說明(H)
收音機 音樂 媒體導覽
節目:
短片:
作者:
著作權:
正在播放
00:22 / 596:31:23

video.rtsp - Windows Media Player
檔案(F) 檢視(V) 播放(P) 我的最愛(A) 移至(G)
說明(H)
收音機 音樂 媒體導覽
節目:
短片:
作者:
著作權:
正在播放
00:06 / 03:43

Figure 46 – Using Windows® Media Player® as Clients. The left one is for live-captured while the right one is for stored video.

6.4 Observations on Server Transmission Behavior

<u>Inter-departure Time between Groups of Packets</u>

In the proposed server system, each media frame is segmented into a group of RTP packets. Therefore, for the video with the framerate of 25 frames per second, supposedly the inter-departure time between RTP packet groups should be 40 milliseconds. To guarantee that the inter-departure times are accurate, in the system, the actual control of the timing is done by reading the system clock. This is as far as what can be done by the from the software point of view. To verify that the resulted inter-departure times match the expected values, some measurements are taken and presented in this section. To conduct this simple experiment, the software “WinPcap: The Windows Packet Capture Library” [22] is utilized to measure the departure times of all packets.

In Figure 47, the histogram of inter-departure times between groups of packets when the server is streaming to a single client is shown. Note that the horizontal axis represents the inter-departure times between groups of packets, while the vertical axis represents the accumulated numbers of times for different inter-departure times. As can be observed from the figure, the measured inter-departure times between groups are not 40 milliseconds. Instead, the bars in the histogram mainly appear at three locations: 23, 35, and 47 milliseconds. Referring to Figure 48, when the situation becomes more complicated (i.e. more clients are connecting to the server), this phenomenon still exist. Note that even though the bars still mainly appear at the three mentioned locations, however, they do spread out noticeably.

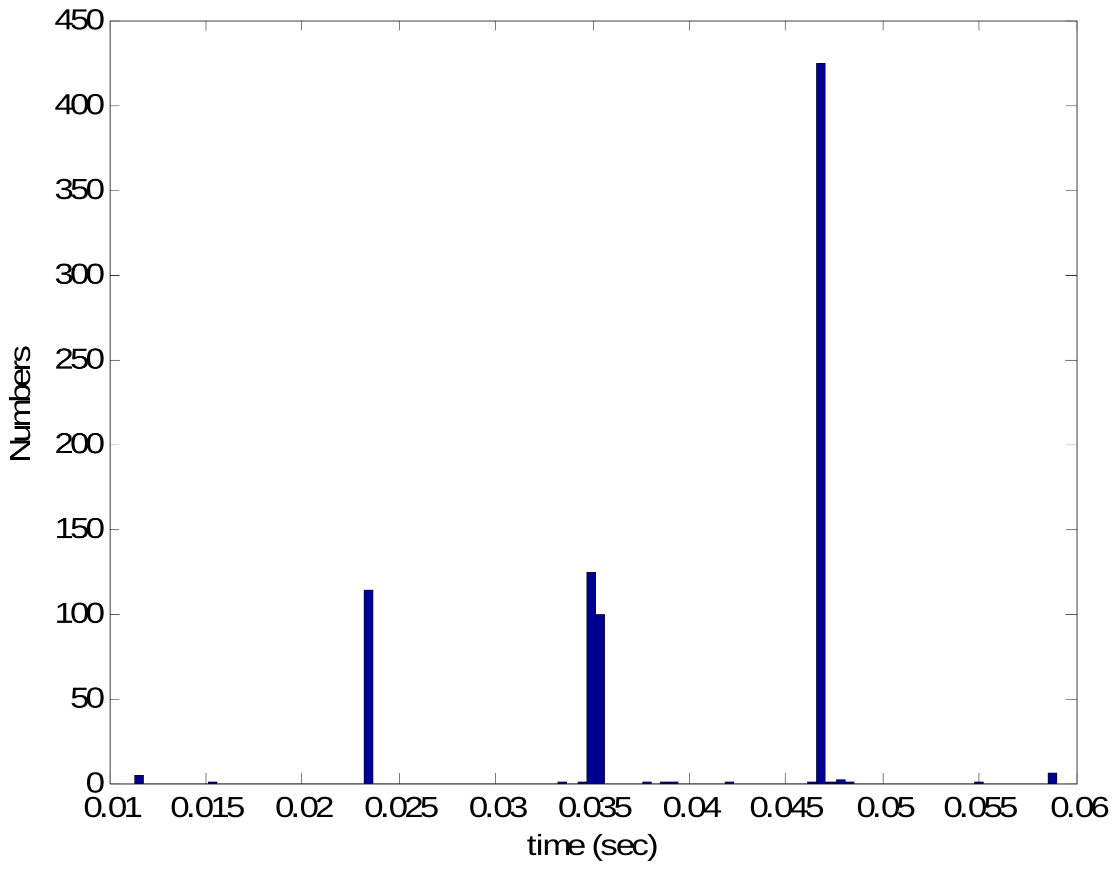


Figure 47 – Histogram of inter-departure times between groups of packets when streaming to one client

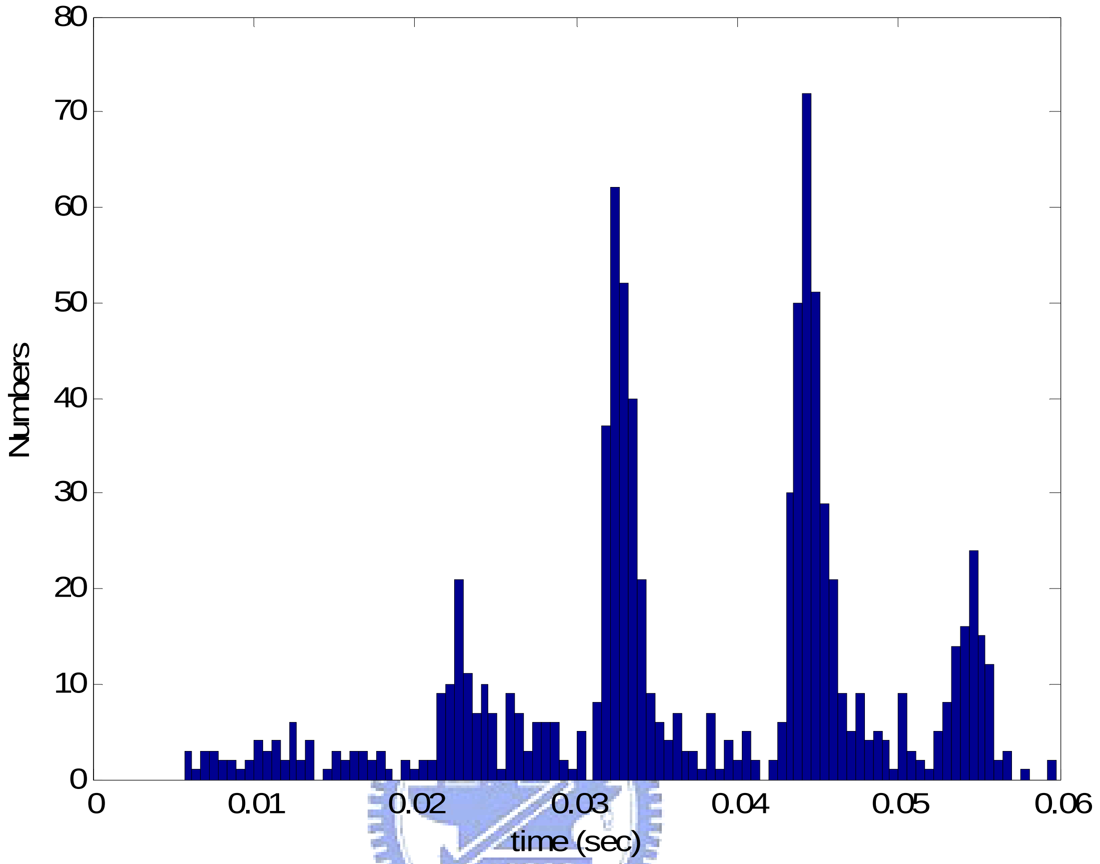
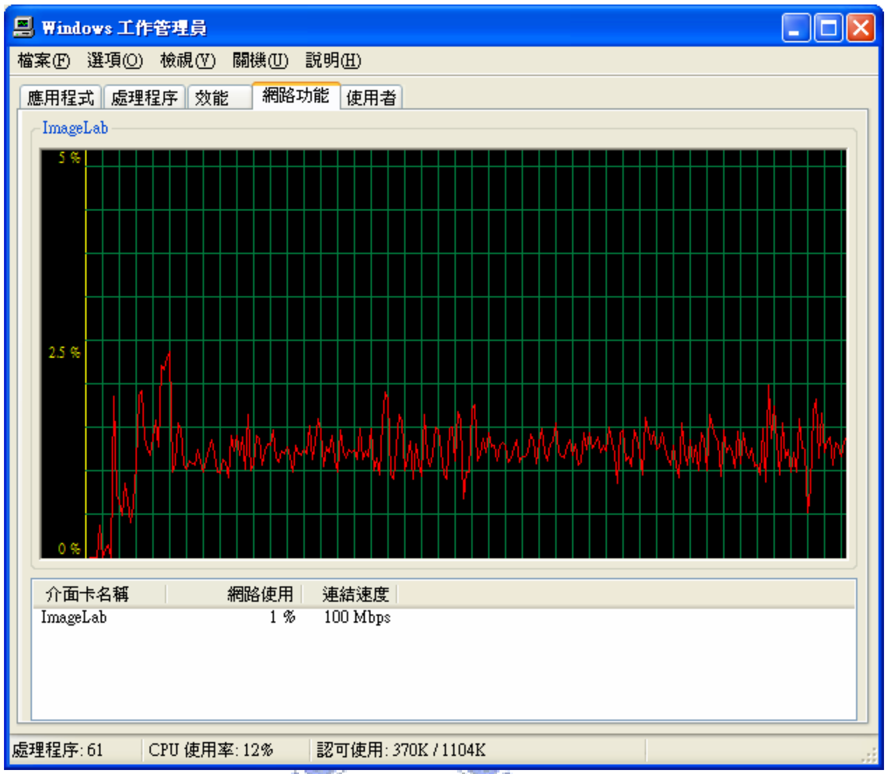


Figure 48 – Histogram of inter-departure times between groups of packets when streaming to six clients

Bandwidth Smoothing Results

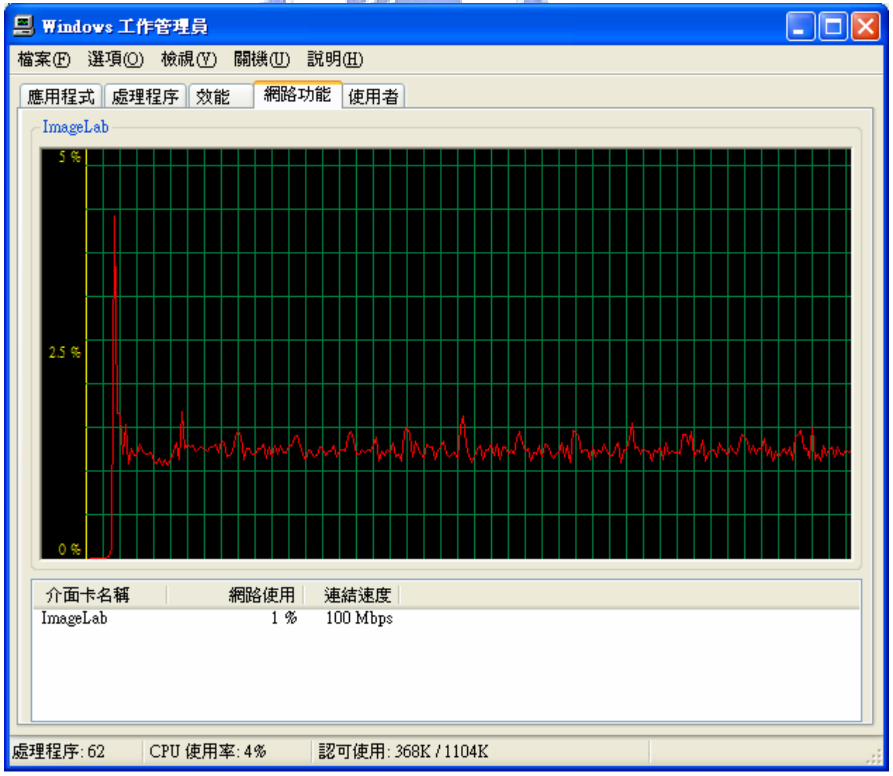
In the proposed server system, the user can select whether to turn on the packet-based MVBA traffic smoothing feature or not. To test the performance of this feature, the networking monitoring function of Windows® Task Manager is employed. As can be observed from Figure 49, when this feature is not in use, the traffic is bursty due to the VBR characteristic of MPEG-4 compression algorithm. However, when the smoothing feature is enabled, the traffic is much smoother as shown by Figure 50. Note that only 2 MB client buffer size is assumed when computing the smoother transmission plan. In fact, the client buffer size is normally larger.


Windows 工作管理員
檔案(F) 選項(O) 檢視(V) 關機(U) 說明(H)
應用程式 處理程序 效能 網路功能 使用者
ImageLab
5 %
2.5 %
0 %

介面卡名稱	網路使用	連結速度
ImageLab	1 %	100 Mbps

處理程序: 61 CPU 使用率: 12% 認可使用: 370K / 1104K

Figure 49 – Transmission rate for unsmoothed traffic


Windows 工作管理員
檔案(F) 選項(O) 檢視(V) 關機(U) 說明(H)
應用程式 處理程序 效能 網路功能 使用者
ImageLab
5 %
2.5 %
0 %

介面卡名稱	網路使用	連結速度
ImageLab	1 %	100 Mbps

處理程序: 62 CPU 使用率: 4% 認可使用: 368K / 1104K

Figure 50 – Transmission rate for smoothed traffic with 2MB client buffer

6.5 Discussions

Regarding to the execution results, it is proud to say that the server and the client systems work successfully. They both meet the proposed expectations. In particular, the server system can acquire audiovisual contents from various types of cameras. In addition, it can transcode these acquired contents into MPEG-4 format in real-time. It can provide streaming services to multiple clients, who may ask for either live-captured contents or stored media files. The data transmission can take place either via UDP or TCP, as designated by the client. If the desired media file is stored type, the server may stream it in a smoother manner since the bandwidth smoothing technique is embedded into the server. Finally, the server is fully compliant with both RTP and RTSP protocols. For the client side, the client system can connect to the server to request the desired media contents. Upon receiving MPEG-4 data, it can decode and render them in real-time. Lastly, the proposed system is so flexible that the user can even use Windows® Media Player® as the client program.

The implemented client and server system seem to work fine until the measurements mentioned above are taken. The resulted inter-departure times are not the same as those of expected when the server is streaming to a single client. This problem gets worse when the server is streaming to multiple clients as described earlier. The possible reasons for this discrepancy are:

- The TCP and the UDP socket kernel programs (Windows® Socket Library) do not function in the expected way. This is to say that when "*send()*" or "*sendto()*" is called to send a packet via TCP or UDP, respectively, the packet is not transmitted immediately.
- The bars in the histogram of second case (Figure 48) spread out because:
  - Socket kernel programs are too busy to handle outbound packets destined to six clients. Therefore, packets cannot be injected into the network on time.
  - CPU is exhausted due to fact that there are too many packetizers ("*StreamerletIs()*" threads) executing concurrently, and as a result, "*send()*" or "*sendto()*" functions are not called at the exact times.

This problem may become more serious when the wireless transmission comes into play whether at the client side or the server side because:

- IEEE 802.11 DCF mechanism will cause one’s transmission delayed when the medium is in use. Thus, the inaccuracy of inter-departure times will probably be enlarged.
- When multiple stations are sharing the same medium (i.e. air), the overall transmission rate may eventually saturate [23], and cannot reach the maximum supported transmission rate of IEEE 802.11. Consequently, the delayed inter-departure times might be extended even longer.

Finally, the packet-based MVBA algorithm is successfully embedded into the server system. This feature can be useful when streaming stored media files. However, since MVBA is not intended for online or real-time applications, such as sports events or video conferences, respectively, this feature cannot be used to improve the transmission of live-captured audiovisual contents.

7 Conclusions

To reduce the time that people have to wait before starting to enjoy the multimedia contents, streaming technology has been developed and widely accepted. The mainstream streaming protocols are RTP and RTSP. The former is designed for the needs of streaming data with real-time characteristics. It has a subpart named RTCP, which allows the monitoring of reception quality. The later acts like a “network remote control”. It provides bi-directional communication between the server and the client. Many commercial streaming solutions that adopt these two protocols are out there. However, they are incapable of getting media contents from various types of cameras, and smoothing bursty traffic.

In this work, a real-time interactive RTP/RTSP multimedia streaming monitoring system with bandwidth smoothing technique is designed, developed and implemented successfully. In short, unlike other commercial solutions that merely can stream stored video or audio, the proposed system can stream either live-captured audiovisual contents that are real-time encoded into MPEG-4 format or stored ones.

Apparently, the server system is capable of acquiring captured images from various types of cameras. In addition, to smoothen the burstiness of VBR contents that may cause inefficient utilization of the network resources, the proposed system is further enhanced by the packet-based MVBA smoothing algorithm, which can compute a smoother transmission plan. Moreover, in the client system, a simple and easy image capturing software interface is designed and implemented. With this interface, one can acquire received media frames effortlessly, so he or she can do any kind of post-processing. For example, if automotive image processing functions are needed to allow automatic monitoring on unattended target zones, the system developer can get received frames from the image capturing software interface without any difficulty, and in turn feed them to the image processing functions in a graceful way.

Finally, the execution results are presented. These results indicate that the proposed system is implemented successfully, and that the goals of this work are achieved. However, some issues do arise. The inter-departure times are not equal to the expected values. This discrepancy becomes more obvious when more clients are present. Without a doubt, if IEEE 802.11 wireless connection is in use either at the client side or the server side, this problem will be even more harmful.

From the system designer’s point of view, let’s conclude what factor limits the capability of the server system. If the main threads used by the server are to be divided into groups based on their functions, they can be sorted into the following units:

- Transcoding unit
  - It includes “*Transcoder()*” thread.
  - The server system only contains a single entity of this unit.
  - It is responsible for acquiring images from various types of cameras, transcoding these images into MPEG-4, and placing the resulted bitstreams onto the buffer where the planning unit can access.
  - The number of cameras that this unit can support to perform real-time transcoding depends on the computation power of the server machine.
- Transmission unit
  - It includes “*PacketScheduler()*” thread.
  - The server system only contains a single entity of this unit.
  - It is responsible for handing over all the RTP packets destined to each connected clients to the network card, which in turn inserts these packets to the network.
- Planning unit
  - It includes “*StreamerletIs()*” and “*RTSPProcessorThread()*” threads.
  - Whenever the server accepts a new connection from a client, the server will exclusively create a new entity of this unit to deal with all the RTP and RTSP packets destined to or received from this client.
  - “*StreamerletIs()*” is in charge of packetizing frames acquired either from the storage device or the buffer where the transcoding unit places the MPEG-4 bitstreams into RTP packets, and putting these packets on the buffer where the

transmission unit can access.

- "*RTSPProcessorThread()*" is in charge of exchanging RTSP messages with the client and performing necessary RTSP state transitions.

Obviously, among these three units, the most CPU-consuming one is the transcoding unit. This is because of the fact that the MPEG-4 encoding task requires large computation power, especially its motion search part, which accounts for over eighty percent of the overall MPEG-4 computation. In other words, if the transcoding unit is not taken into consideration, then the server can easily stream media data to more than ten clients concurrently. The reasons are that the transmission unit in fact consumes light computation power, and even though the number of the entities of the planning units increases as the number of the clients becomes larger, but due to their low computational power requirement, the overall server system performance is not affected greatly. Therefore, the key factor that limits the server's capability is the encoding unit.

After numerous measurements are taken, it is found that 7 milliseconds in average will be needed by the MPEG-4 encoder to encode a single 320x240 full-color frame. For the case when images acquired from a single camera are to be encoded at the frame rate of 25 frames per second, which is equivalent to 40 milliseconds of inter-frame time, then within this period of time the server system will take about 10 milliseconds to perform transcoding, and 1 millisecond to transmit a group of RTP packets (around 5 to 6 packets) to each connected client. Thus, in this case, the server can support up to 30 clients simultaneously. If the server provides the service of streaming three different videos acquired from three cameras to clients, it will take 30 milliseconds out of the 40-millisecond inter-frame time just to transcode the captured contents into MPEG-4. As a result, the number of clients that the server can support is decreased to 10.

In the future, the system may be enhanced by adding the following features to it:

- The feedback mechanism provided by RTCP can be implemented to the system so the server can adapt its transmission behavior to the network conditions.
- The proposed packet-based MVBA smoothing algorithm is designed for stored

media applications and cannot apply to the live-events such as live baseball games (i.e. small start-up delay is tolerable). Therefore, online smoothing techniques should be studied and realized in the future system.

- For some applications, any noticeable delay is not allowed. Thus, real-time smoothing algorithms should be researched and implemented in the upcoming system.
- Since RTSP is extendible, some other nice functions should be added to the system. For example, for the client system that contains automotive image processing functions, if RTSP is modified to provide a means for the client to send messages to the server, then when the client detects that the monitored area is in danger, it can report to the server about the abnormal event via RTSP channel. In turn, the server can either record the event or notify the related personnel.
- Since the current transcoding unit is CPU-consuming, the server system can be improved by either upgrading the CPU or replacing the encoder with a more efficient one.

References

[1] Jan Krikke, "Streaming video transforms the media industry", IEEE Computer Graphics and Applications, Volume 24, Issue 4, July/Aug, 2004.

[2] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", Audio Visual Working Group Request for Comment RFC 3550, IETF, July 2003.

[3] H. Schulzrinne, A. Rao, R. Lanphier, M. Westerlund, and A. Naraismhan, "Real time streaming protocol (RTSP)", Internet Draft RFC 2326, Internet Engineering Task Force, October 27, 2003.

[4] M. Handley and V. Jacobson, "SDP: Session Description Protocol", Request for Comments: 2327, Internet Engineering Task Force, April, 1998.

[5] D. Anderson, Y. Osawa, and R. Govindan, "A file system for continuous media," ACM Trans. Comput. Syst., pp. 311–337, Nov. 1992.

[6] P. Lougher and D. Shepard, "The design of a storage server for continuous media," Comput. J., vol. 36, pp. 32–42, Feb. 1993.

[7] D. Gemmell, J. Vin, D. Kandlur, P. Rangan, and L. Rowe, "Multimedia storage servers: A tutorial," IEEE Comput. Mag., vol. 28, pp. 40–49, May 1995.

[8] I. Dalgic and F. A. Tobagi, "Performance evaluation of ATM networks carrying constant and variable bit-rate video traffic," IEEE J. Select. Areas Commun., vol. 15, pp. 1115–1131, Aug. 1997.

[9] T. V. Lakshman, A. Ortega, and A. R. Reibman, "Variable bitrate (VBR) video: Tradeoffs and potentials," Proc. IEEE, vol. 86, pp. 952–973, May 1998.

[10] O. Rose, "Statistical properties of MPEG video traffic and their impact on traffic modeling in ATM systems," in Proc. Conf. Local Computer Networks, Oct. 1995, pp. 397–406.

[11] Wu-chi Feng and Jennifer Rexford," Performance Evaluation of Smoothing Algorithms for Transmitting Prerecorded Variable-Bit-Rate Video ", in IEEE Transactions on Multimedia, vol. 1, NO. 3, September 1999.

[12] W. Feng and S. Sechrest, "Smoothing and buffering for delivery of prerecorded compressed video," in Proc. of the IS&T/ SPIE Symposium on Multimedia Comp. and Networking, pp. 234-242, Feb. 1995.

[13] J. D. Salehi, Z.-L. Zhang, J. F. Kurose, and D. Towsley, "Supporting stored video:

Reducing rate variability and end-to-end resource requirements through optimal smoothing," IEEE/ACM Trans. Networking, vol. 6, pp. 397–410, Aug. 1998.

[14] W. Feng, F. Jahanian, and S. Sechrest, "Optimal buffering for the delivery of compressed prerecorded video," ACM Multimedia System Journals, pp. 297–309, Sept. 1997.

[15] W. Feng, "Rate-constrained bandwidth smoothing for the delivery of stored video," in Proc. IS&T/SPIE Multimedia Networking and Computing, Feb. 1997, pp. 58–66.

[16] J. M. McManus and K. W. Ross, "Video on demand over ATM: Constant-rate transmission and transport," IEEE J. Selected Areas Communications, vol. 14, pp. 1087–1098, Aug. 1996.

[17] Jean M. McManus, Keith W. Ross, "A dynamic programming methodology for managing prerecorded VBR sources in packet-switched networks," Telecommunication Systems, vol. 9, 1998.

[18] J. Zhang and J. Hui, "Traffic characteristics and smoothness criteria in VBR video transmission," in Proc. IEEE Int. Conf. Multimedia Computing and Systems, June 1997.

[19] "DLLs, Processes, and Threads." available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/dlls_processes_and_threads.asp.

[20] W. Richard Stevens, "Unix Network Programming, Networking APIs: Sockets and XTI", Prentice Hall PTR, vol. 1, 2nd Edition, pp. 44－46, 1998.

[21] Ferguson P., Huston G., Quality of Service: Delivering QoS on the Internet and in Corporate Networks, John Wiley & Sons, Inc., 1998.

[22] "WinPcap, the Packet Capture and Network Monitoring Library for Windows" available at http://www.winpcap.org/default.htm.

[23] Shihong Zou, Haitao Wu, and Shiduan Cheng, "A New Mechanism of Transmitting MPEG-4 Video in IEEE 802.11 Wireless LAN with DCF", on IEEE Proceedings of ICCT, 2003.

[24] Chan-Wei Lin, "Smoothing Algorithm for Video Streaming in Packet Based Transmission System", Master Thesis, The Department of Communication Engineering, National Chiao Tung University.