

國立交通大學

電信工程學系

碩士論文

串流平台的程序控制

**Process Control in Streaming Server**

研究生：邱程翔

指導教授：張文鐘 博士

中華民國九十六年八月

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 and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

August 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月

# 串流平台的程序控制

研究生：邱程翔

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

串流伺服器的程序控制主要包含了連線建立、檔案處理及封包傳送三項主要任務。連線建立包含了 socket 間的資料傳遞以及透過一個狀態的機制來傳遞信令；檔案處理便是影音格式的解析以及訊框的切割與封裝；最後的部份便是封包傳送，此部分的重點在於時間的控制機制。

本論文主要是從單行程的觀點去探討串流伺服器的建構程序。一般來說，串流伺服器可分為多行程以及單行程架構，這兩種架構在表現上雖然差別不明顯，但是單行程細部的時間控制相較於多行程便困難許多，因為這牽涉到了影音格式的解編碼問題。

此外，本論文也實際修改一套開程式碼的串流信令，使得改良後的串流系統能夠和市面上的商業串流系統得以溝通，其細部的修改重點也是本論文論述的核心所在。

# **Process Control in Streaming Server**

Student : Cheng - Siang Chiu    Advisor : Dr. Wen – Thong Chang

Department of Communication Engineering  
National Chiao Tung University

## **Abstract**

Streaming systems consist of three big jobs which include connection setup, file processing, and packets sending. Connection setup consists of the socket setup and the state mechanism which is used to exchange signaling; file processing contains the file formats' parser and the packetizer; packets sending focuses on the timing control mechanism.

In this work, we investigate how streaming servers are constructed in a single process's view. Generally speaking, streaming servers can be categorized into two structures which are multi process structure and single process structure. Although the differences in the performance of the two structures are not obvious, the detail implementations of single process are more complicated in timing control since the coding problems

of the file formats.

Besides, we also try to modify an open source streaming server so that it can connect to commercial streaming servers by RTSP signaling.



## 致謝

感謝論文指導教授 張文鐘 博士，在兩年的讀書期間，不斷磨練、教導我，不論颶風下雨，也會在實驗室和我們互相討論及研究學問，還有花時間指點論文不足的地方，再次謝謝老師兩年的教誨。同時也感謝 林大衛教授、廖維國教授及何文楨主任，於口試時的寶貴建議。

感謝實驗室一起奮鬥的學長：培哲學長、為棟學長，同學：宗修、新華、榮勝、書維、政鴻、瑩甄、孟潔以及素仙，學弟：宗學、文賢、明山、政達以及峻權，特別謝謝幫我看程式、整理數據和陪我聊天的素仙，和大家一起讀書、討論功課、打球，讓我這兩年過得特別充實，也特別具有懷念的價值。

此外，還有一起住了兩年的室友，博元、舜庭、堉棋、憲諒、亭州以及建男，懷念和你們一起吃東西聊天的時光。

最後，感謝我的家人，爸爸、媽媽以及老弟，在這兩年給我的鼓勵以及支持，謝謝您們。

## **Acknowledgement**

I would like to express my gratitude to my academic and research advisor Dr. Wen-Thong Chang for his guidance and constant support in helping my 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.



I owe my sincere appreciation to my families and relatives who have supported and encouraged me over the years. 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

摘要 .....	I
ABSTRACT.....	II
致謝 .....	IV
ACKNOWLEDGEMENT.....	V
TABLE OF CONTENTS .....	VI
LIST OF FIGURES.....	VIII
LIST OF TABLES.....	IX
CHAPTER 1.....	1
1.1 BACKGROUND.....	1
1.2 MOTIVATIONS .....	4
1.3 RESEARCH GOALS .....	4
1.4 THESIS OUTLINES .....	5
CHAPTER 2 .....	6
2.1 THE REAL-TIME TRANSPORT PROTOCOL.....	6
2.1.1 RTP.....	7
2.1.2 RTCP.....	9
2.1.3 RTSP.....	10
2.2 INTRODUCTION TO FFSERVER .....	16
2.3 MODIFICATIONS IN FFSERVER.....	20
CHAPTER 3 .....	25
SYSTEM ARCHITECTURE AND FLOWCHART OF .....	25
3.1 SYSTEM ARCHITECTURE OF A STREAMING SERVER.....	25
3.1.1 SOCKET SETUP ( Code Segment A ) In.....	29
A Streaming Server .....	29
3.1.2 LABELING ( Code Segment B ) In A Streaming Server ..	32
3.1.3 SELECT ( Code Segment C ) In A Streaming.....	35
Server.....	35
3.1.4 HANDLING ( Code Segment D ) In FFserver .....	43
3.1.5 ACCEPT ( Code Segment E ) In FFserver.....	47
3.1.6 Overall Procedures In FFserver .....	49
3.2 RTSP METHODS IMPLEMENTED IN FFSERVER .....	52
3.2.1 RTSP_CMD_OPTIONS().....	52



<b>3.2.2</b>	<b><i>RTSP_CMD_DESCRIBE()</i></b> .....	53
<b>3.2.3</b>	<b><i>RTSP_CMD_SETUP()</i></b> .....	56
<b>3.2.4</b>	<b><i>RTSP_CMD_PLAY()</i></b> .....	58
<b>CHAPTER 4</b>	.....	<b>61</b>
<b>4.1</b>	<b>RESULTS BEFORE AND AFTER THE MODIFICATIONS IN FFSERVER</b> ....	<b>61</b>
<b>4.2</b>	<b>PROCEDURE OF CODE SEGMENTS</b> .....	<b>64</b>
<b>CHAPTER 5</b>	.....	<b>66</b>
<b>REFERENCES</b>	.....	<b>67</b>
<b>APPENDIX</b>	.....	<b>68</b>



## List of Figures

Figure 2- 1	The main structure of FFmpeg .....	18
Figure 2- 2	Results of connection to FFserver .....	19
Figure 2- 3	Code segment 1 needs to be modified.....	21
Figure 2- 4	Code segment 2 needs to be modified.....	21
Figure 2- 5	Code segment 3 needs to be modified.....	23
Figure 2- 6	Code segment 4 needs to be modified.....	23
Figure 3- 1	System Architecture of FFserver.....	25
Figure 3- 2	Pseudo Codes of http_server() in FFserver.....	26
Figure 3- 3	Components of SOCKET SETUP .....	31
Figure 3- 4	Flowchart of Code Segment B .....	34
Figure 3- 5	Four macros of fd_set datatype .....	38
Figure 3- 6	Example of how fd_set is constructed .....	38
Figure 3- 7	Pseudo Codes of Code Segment C.....	41
Figure 3- 8	Flowchart of Code Segment C .....	42
Figure 3- 9	Flowchart of Code Segment D.....	45
Figure 3- 10	Pseudo Codes of Code Segment D.....	46
Figure 3- 11	Pseudo Codes of rtsp_parse_request() .....	46
Figure 3- 12	Flowchart of Code Segment E .....	48
Figure 3- 13	Pseudo Codes of rtsp_cmd_options().....	53
Figure 3- 14	Pseudo Codes of rtsp_cmd_describe().....	54
Figure 3- 15	Pseudo Codes of prepare_sdp_description().....	54
Figure 3- 16	Pseudo Codes of rtsp_cmd_setup().....	56
Figure 3- 17	Pseudo Codes of rtsp_cmd_play() .....	58
Figure 3- 18	PTS : the pictures' display order .....	60
Figure 3- 19	DTS : the pictures' coding order .....	60
Figure 4- 1	Old ffserver's packets captured by Ethereal .....	61
Figure 4- 2	Packets after first modifications.....	62
Figure 4- 3	All modifications are completed .....	63
Figure 4- 4	Procedure of code segments.....	64

## List of Tables

Table 2- 1	Overview of RTSP methods, their direction, and what objects they operate on. ....	15
Table 3- 1	Payload Types Constructed In FFserver .....	55



# CHAPTER 1

## INTRODUCTION

### 1.1 Background

With the rising capacity of bandwidth and the increasing processing speed of CPUs, we can transmit more multimedia audio/video data, not just texts or pictures over the internet. Therefore, the applications of multimedia have played an unreplaceable role in our daily life. However, even with the current capacity of bandwidth, usually 10Mbps downloading speed, it is still a time-consuming job to download a movie that is one and half an hour long, 1GB in size, when users want to see movies online. Because it costs at least 15 to 20 minutes to download the whole movie if the network is not congested, and users usually can not stand waiting so much long time. Users may lose their interests over watching movie when the network is congested or the movie is not the one they want to see after the movie is downloaded.

To overcome this problem, the new 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 users. The major distinction between streaming and the traditional downloading is that the former offers “play while downloading” while the later only supports “play after downloading.” With streaming users can watching movies while downloading them.

There are three important protocols used in streaming. They are The Real-Time Streaming Protocol (RTSP), Real-Time Transport Protocol (RTP), and the Real-Time Transport Control Protocol (RTCP). They were specifically designed to stream media over networks. The latter two are built on top of User Datagram Protocol (UDP). RTP defines a standardized packet format for delivering audio and video over the internet. And the other two counterparts, RTCP and RTSP, work with RTP to make streaming systems more reliable. More details about the three protocols are presented in Chapter 2.

UDP sends the media stream as s series of small packets. This is simple and efficient; however, packets are liable to be lost or corrupted in transit. Depending on the protocol and the extent of the loss, the user may be able to recover the data with error correction techniques, may interpolate over the missing data, or may suffer a dropout.

Transmission Control Protocol (TCP) guarantees correct delivery of each bit in the media stream. However, they accomplish this with a system of timeouts and retries, which makes them more complex to implement. It also means that when there is data loss on the network, the media stream stalls while the protocol handlers detect the loss and retransmit the missing data. Users can minimize the effect of this by buffering data for display.

In any multimedia system, a powerful streaming server is the key component to judge if the system is an excellent competitor in the market. However, in the modern market three mainstream streaming servers, Media Services, RealSystem and Quick Time Server, could not talk to each other by using RTSP. Each company does not release its server's source codes and roughly follow the standards. That results in a circumstance that a player could not connect to a rival streaming server. It is the situation that confuses every consumer. Because it means consumers have to use a set of multimedia systems developed by a single company. So in this research we try to modify an open source streaming server, FFserver, to be able to stream multimedia data to another open source player, VLC media player.

## 1.2 Motivations

The motivations of this work are as the following :

- The open source FFserver can not support RTSP signaling very well.
- Timing control is one of the most critical issue in any streaming system.
- Single process streaming server is hard to construct.



## 1.3 Research Goals

The goals of this research are as follows:

- Modifying a streaming server so that it could stream multimedia data by RTSP.
- Figuring out which parameters or syntax are necessary in the RTSP.
- Describing one feasible timing control algorithm implemented in streaming servers.

## 1.4 Thesis Outlines

The organization of this thesis is as follows. Firstly, a brief background of know-how and some code modifications in FFserver are presented in Chapter 2. Secondly, the structures and procedures of an open source streaming server are covered in Chapter 3. In Chapter 4, experimental results are the main topics. At last, research conclusions and future works are depicted in Chapter 5.





# **CHAPTER 2**

## **STREAMING TECHNOLOGIES**

### **AND**

## **FFSERVER'S INTRODUCTION**

In this chapter, we present some important technologies and FFserver's introduction. We briefly introduce The Real-Time Transport Protocol (RTP) [2], The Real-Time Transport Control Protocol (RTCP) [2], and The Real-Time Streaming Protocol (RTSP) [3] firstly. Then FFserver is the following discussions which cover socket and some modifications in FFserver.



### **2.1 The Real-Time Transport Protocol**

The Transmission Control Protocol (TCP) is one of the core protocols in the internet protocol suite. Using TCP, applications on networked hosts can create connections to one another, over which they can exchange streams of data using Stream Sockets. The protocol guarantees reliable and in-order delivery of data from sender to receiver. That means retransmissions of packets are performed when packets are missed or timed out.

Consequently, TCP does not assure the timely delivery of packets.

Despite the fact that the mechanisms of TCP are necessary in many applications, such as World Wide Web, e-mail and File Transfer Protocol, real-time transmissions can not use it. Another protocol, The Datagram Protocol (UDP), can solve the timely delivery problem. Since UDP does not support retransmission, it provides a faster transmission than TCP dose over congested networks.

The Real-Time Transport Protocol (RTP) has the ability to compensate for the missing functions of UDP. And RTP is usually based on top of UDP; that is, RTP packets are usually encapsulated in UDP packets. In the following three sections, we describe the fundamental concepts of RTP, The Real- Time Transport Control Protocol (RTCP) and The Real-Time Streaming Protocol (RTSP).

### **2.1.1 RTP**

RTP defines a standardized packet format for delivering audio and video over the internet. It was developed by the Audio-Video Transport Working Group

of the IETF and first published in 1996 as RFC 1889 which was made obsolete in 2003 by RFC 3550. RTP can carry any data with real-time characteristics, such as interactive audio and video. It goes along with the RTCP and it's built on top of UDP. Applications using RTP are less sensitive to packet loss, but typically very sensitive to delays, so UDP is a better choice than TCP.

According to RFC 1889, RTP provides the following services :

- Payload-type identification – Indication of what kind of content is being carried.
- Sequencing numbering – PDU sequence numbering.
- Time stamping – allow synchronization and jitter calculations.
- Delivery monitoring.

### **2.1.2 RTCP**

The Real-Time Transport Control Protocol (RTCP), defined in RFC 3550 [2], is a sister protocol of RTP. RTCP provides out-of-band control information for RTP in the delivery and packaging of multimedia data, but does not transport any data itself. It is used periodically to transmit control packets to participants in a streaming multimedia session. The primary function of RTCP is to provide feedback on the Quality of Service (QoS) being provided by RTP.

RTCP gathers statistics on a media connection and information such as bytes sent, packets sent, lost packets, jitter, feedback and round trip delay. An application may use this information to increase the quality of service perhaps by limiting flow, or maybe using a low compression codec instead of a high compression codec. RTCP is used for QoS reporting.

There are several type of RTCP packets: Sender report packet, Receiver report packet, Source Description RTCP Packet, Goodbye RTCP Packet and Application Specific RTCP packets.

### **2.1.3 RTSP**

The Real Time Streaming Protocol (RTSP), developed by the IETF and created in 1998 as RFC 2326 [3], is a protocol for use in streaming media systems which allows a client to remotely control a streaming media server, issuing VCR-like commands such as "play" and "pause", and allowing time-based access to files on a server.

The protocol is similar in syntax and operation to HTTP but RTSP adds new requests. A summary of available RTSP methods are listed in Table 2.1. And the followings are the brief introductions of each method.

#### **➤ DESCRIBE**

- The DESCRIBE method retrieves the description of a presentation or media object identified by the request URL from a server. It may use the Accept header to specify the description formats that the client understands. The server responds with a description of the requested resource. The DESCRIBE reply-response pair constitutes the media initialization phase of RTSP.

➤ **ANNOUNCE**

- The ANNOUNCE method serves two purposes :
  - i. When sent from client to server, ANNOUNCE posts the description of a presentation or media object identified by the request URL to a server.
  - ii. When sent from server to client, ANNOUNCE updates the session description in real-time.

➤ **GET\_PARAMETER**

- The GET\_PARAMETER request retrieves the value of a parameter of a presentation or stream specified in the URI. The content of the reply and response is left to the implementation. GET\_PARAMETER with no entity body may be used to test client or server liveness (“ping”).

➤ **OPTIONS**

- The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or

initiating a resource retrieval.

➤ **PAUSE**

- The PAUSE request causes the stream delivery to be interrupted (halted) temporarily. If the request URL names a stream, only playback and recording of that stream is halted. For example, for audio, this is equivalent to muting. If the request URL names a presentation or group of streams, delivery of all currently active streams within the presentation or group is halted. After resuming playback or recording, synchronization of the tracks **MUST** be maintained. Any server resources are kept, though servers **MAY** close the session and free resources after being paused for the duration specified with the timeout parameter of the Session header in the SETUP message.

➤ **PLAY**

- The PLAY method tells the server to start sending data via the mechanism specified in SETUP. A client **MUST NOT** issue a PLAY request until any outstanding SETUP requests have been acknowledged as successful. The PLAY request positions the normal play time to the beginning of the range specified and delivers stream data until

the end of the range is reached.

➤ **RECORD**

- This method initiates recoding a range of media data according to the presentation description. The timestamp reflects start and end time. If no time range is given, use the start or end time provided in the presentation description. If the session has already started, commence recording immediately.

➤ **REDIRECT**

- A REDIRECT request informs the client that it must connect to another server location. It contains the mandatory header **Location**, Which indicated that the client should issue requests for that URL. It may contain the parameter **Range**, which indicated when the redirection takes effect. If the client wants to continue to send or receive media for this URI, the client **MUST** issue a **TEARDOWN** request for the current session and a **SETUP** for the new session at the designated host.

➤ **SETUP**

- The **SETUP** request for a URI specifies the transport mechanism to be used for the streamed media. A client can issue a **SETUP** request for a stream that is already



playing to change transport parameters, which a server MAY allow. If it does not allow this, it MUST respond with error “455 Method Not Valid In This State”. For the benefit of any intervening firewalls, a client must indicate the transport parameters even if it has no influence over these parameters, for example, where the server advertises a fixed multicast address.

➤ **SET\_PARAMETER**

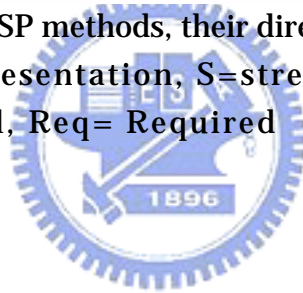
- This method requests to set the value of a parameter for a presentation or stream specified by the URI.

➤ **TEARDOWN**

- The TEARDOWN request stops the stream delivery for the given URI, freeing the resources associated with it. If the URI is the presentation URI for this presentation, any RTSP session identified associated with the session is no longer valid. Unless all transport parameters are defined by the session description, a SETUP request has to be issued before the session can be played again.

method	direction	object	requirement
DESCRIBE	C->S	P,S	recommended
ANNOUNCE	C->S S->C	P,S	optional
GET_PARAMETER	C->S S->C	P,S	optional
OPTIONS	C->S S->C	P,S	required (S->C: optional)
PAUSE	C->S	P,S	recommended
PLAY	C->S	P,S	required
RECORD	C->S	P,S	optional
REDIRECT	S->C	P,S	optional
SETUP	C->S	S	required
SET_PARAMETER	C->S S->C	P,S	optional
TEARDOWN	C->S	P,S	required

Table 2- 1 Overview of RTSP methods, their direction, and what objects they operate on. Legend: P=presentation, S=stream, R=Responds to, Sd=Send, Opt=Optional, Req= Required



## 2.2 Introduction to FFserver

FFserver is the streaming system of the open source software - FFmpeg [1]. We can download FFmpeg from the website [1]. FFmpeg is a system that can record, convert and stream audio and video. It includes five components :

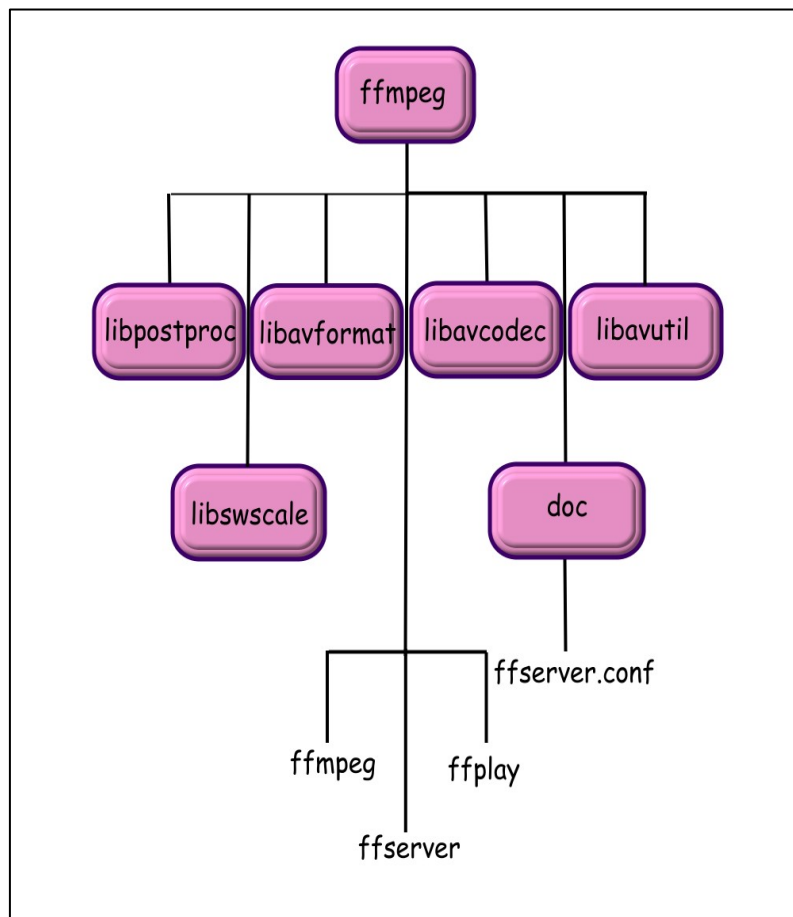
- ffmpeg is a command line tool to convert one file format to another, and it could grab and encode in real time from a TV card.
- ffserver is an HTTP multimedia streaming server. It can't support RTSP connections yet.
- ffmpeg is a media player.
- libavcodec is a library containing all FFmpeg audio/video encoders and decoders.
- libavformat is a library containing parsers and generators for all common audio/video formats.

Since the research is mainly focused on a streaming system, we only discuss FFserver. Just as mentioned above, FFserver is a streaming server that could stream audio/video. But FFserver released on the official website only supports HTTP connections. Therefore, we try to modify FFserver such that it can accept RTSP connections. And we discuss the modifications in section 2.3.

As one of the five components of FFmpeg, FFserver is built under Linux. Hence, we choose to build and modify FFserver under Linux. In the following steps, we show how to build and use FFserver.

- i. Extract the downloaded FFmpeg into one folder. Figure 2-1 is the main architecture of FFmpeg.
- ii. Add the ip address and gateway of client in `/ffmpeg/doc/ffserver.conf` in the form : `ACL allow 140.113.13.245 140.113.13.254`
- iii. Return to `/ffmpeg` folder and type `make` in the command line.

- iv. Type `./ffserver -f doc/ffserver.conf` . Then FFserver is ready to serve.
- v. Open Internet Explorer (IE) and type `140.113.13.247:8090/stat.html` . Figure 2-2 shows the results after the connection is granted.



**Figure 2- 1 The main structure of FFmpeg**

FFServer Status - Microsoft Internet Explorer

檔案(F) 編輯(E) 檢視(V) 我的最愛(A) 工具(T) 說明(H)

網址(O) http://140.113.13.247:8090/stat.html

## FFServer Status

### Available Streams

Path	Served Conns	bytes	Format	Bit rate kbits/s	Video kbits/s	Codec	Audio kbits/s	Codec	Feed
<a href="#">test1.mpg</a>	0	0	mpeg	96	64	mpeg1video	32	mp2	feed1.ffm
<a href="#">test.asf</a>	0	0	asf_stream	256	256	msmpeg4	0		feed1.ffm
<a href="#">file.asf</a>	0	0	asf_stream	33	0	msmpeg4	32	wmav2	/home/file.asf
<a href="#">test1.mpg</a>	0	0	rtp	3384	3000	mpeg2video	384	mp2	/home/test1.mpg
<a href="#">stat.html</a>	1	0	-	-	-	-	-	-	-
<a href="#">index.html</a>	0	0	-	-	-	-	-	-	-

### Feed feed1.ffm

Stream	type	kbits/s	codec	Parameters
0	audio	32	mp2	1 channel(s), 44100 Hz
1	video	64	mpeg1video	160x128, q=3-31, fps=3
2	video	256	msmpeg4	352x240, q=3-31, fps=15

### Connection Status

Number of connections: 1 / 1000  
Bandwidth in use: 0k / 1000k

#	File	IP	Proto	State	Target bits/sec	Actual bits/sec	Bytes transferred
1	stat.html	140.113.13.245	HTTP/1.1	HTTP_WAIT_REQUEST	0	0	0

**Figure 2- 2 Results of connection to FFserver**

## 2.3 Modifications in FFserver

The released version of FFserver on the official web site [1] is not complete in RTSP signaling. Therefore, we can not connect to FFserver by using VLC as a client. So in this section, we try to modify the RTSP signaling of FFserver such that VLC is able to connect to FFserver by RTSP connection. For later discussion's convenience, we dub the modified FFserver new FFserver and the official-released FFserver old FFserver.

We use Wireshark, a packet analyzer, to analyze the packets between old FFserver and VLC. And we found that there are four code sections need to be modified. The followings are the modified segments.

- i. In `ffserver.c`, delete the code segment from line 2745 to line 2750. And this segment is shown in Figure 2-3. Since we do not capture video from live camera, but from the file exits in the disk. We delete this code segment.

```
for(stream=first_stream ; stream!=NULL ; stream=stream->next){  
    if(!stream->is_feed && stream->fmt == &rtp_muxer && !strcmp(path,  
        stream->filename)){  
        goto found;  
    }  
}
```

**Figure 2- 3 Code segment 1 needs to be modified**

- ii. In `ffserver.c`, add the code segment in line 2767. Figure 2-4 presents this code segment. According to RFC 2326 [3] and RFC 2068 [4], this header is required. Content-Base is used to specify the base URI for resolving the relative URLs within the entity.

```
url_fprintf(c->pb,"Content-Base:%s\r\n",url);
```

**Figure 2- 4 Code segment 2 needs to be modified**



iii. In `ffserver.c`, modify the code segment in line 2712. Figure 2-5 exhibits the code segment. If we do not modify the value `i` as `(i+1)%2`, the `PLAY` reply would go wrong. Because the test file in our work is a mpeg file, and there are two streams, video and audio, in this kind of file. After parsing the file, if we do not modify this code segment, audio stream would go along with the statement, " `payload_type=14` and `streamid=0`," and video stream would go along with the statement, " `payload_type=32` and `streamid=1`," in `sdp`. Payload type which is 14 is MPA and 32 is MPV according to Table 3-1 . `Streamid` is used to identify the stream. The request and reply between server and the client do not go wrong. But the contents of `PLAY` method shown on the server are not correct. It shows that audio stream is `streamid 1` and vido stream is `streamid 0`. The contents are not right because we declare `streamid 0` for audio stream and `streamid 1` for video stream in `sdp`. Therefore, we modify the contents in `sdp` as that, `streamid 0` for video stream and `streamid 1` for audio stream. And this time, the contents of `PLAY` method shown on the server are correct, audio stream is `streamid 1` and video stream is `streamid 0`.

```
url_fprintf(pb,"a=control:streamid=%d\n",(i+1)%2);
```

**Figure 2- 5 Code segment 3 needs to be modified**

iv. In Utils.c, delete the code segment from line 2122 to line 2125. And Figure 2-6 shows this code segment. If we do not delete this code segment, we would get the error message “ error non monotone timestamps st->cur\_dts >= pkt->dts st:st->index.” This error message indicates that FFserver does not allow two identical DTS. But if we delete this code segment, FFserver would function very well; that is to say, the statement that there are no two identical DTS is not correct.

```
if(st->cur_dts && st->cur_dts!=AV_NOPTS_VALUE && st->cur_dts >= pkt->dts){  
    av_log(NULL,AV_LOG_ERROR,"error non monotone timestamps  
        %"PRIu64">=%"PRIu64"st:%d\n",st->cur_dts,pkt->dts,st->index);  
    return -1;  
}
```

**Figure 2- 6 Code segment 4 needs to be modified**

After the four code segments are modified, VLC can connect to FFserver by using RTSP signaling connection.



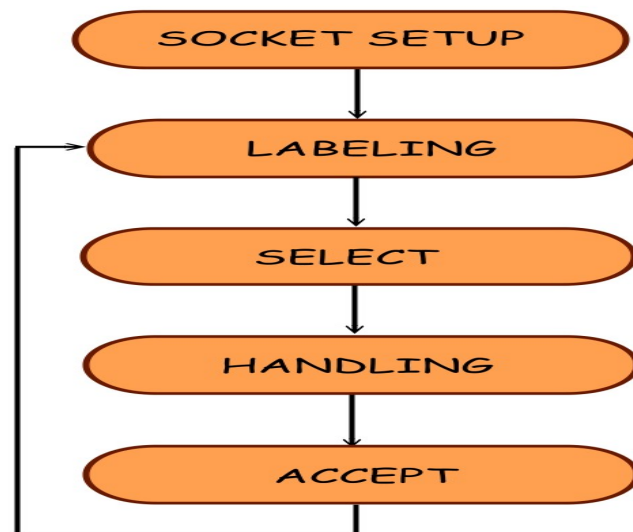
# CHAPTER 3

## SYSTEM ARCHITECTURE AND FLOWCHART OF A STREAMING SERVER

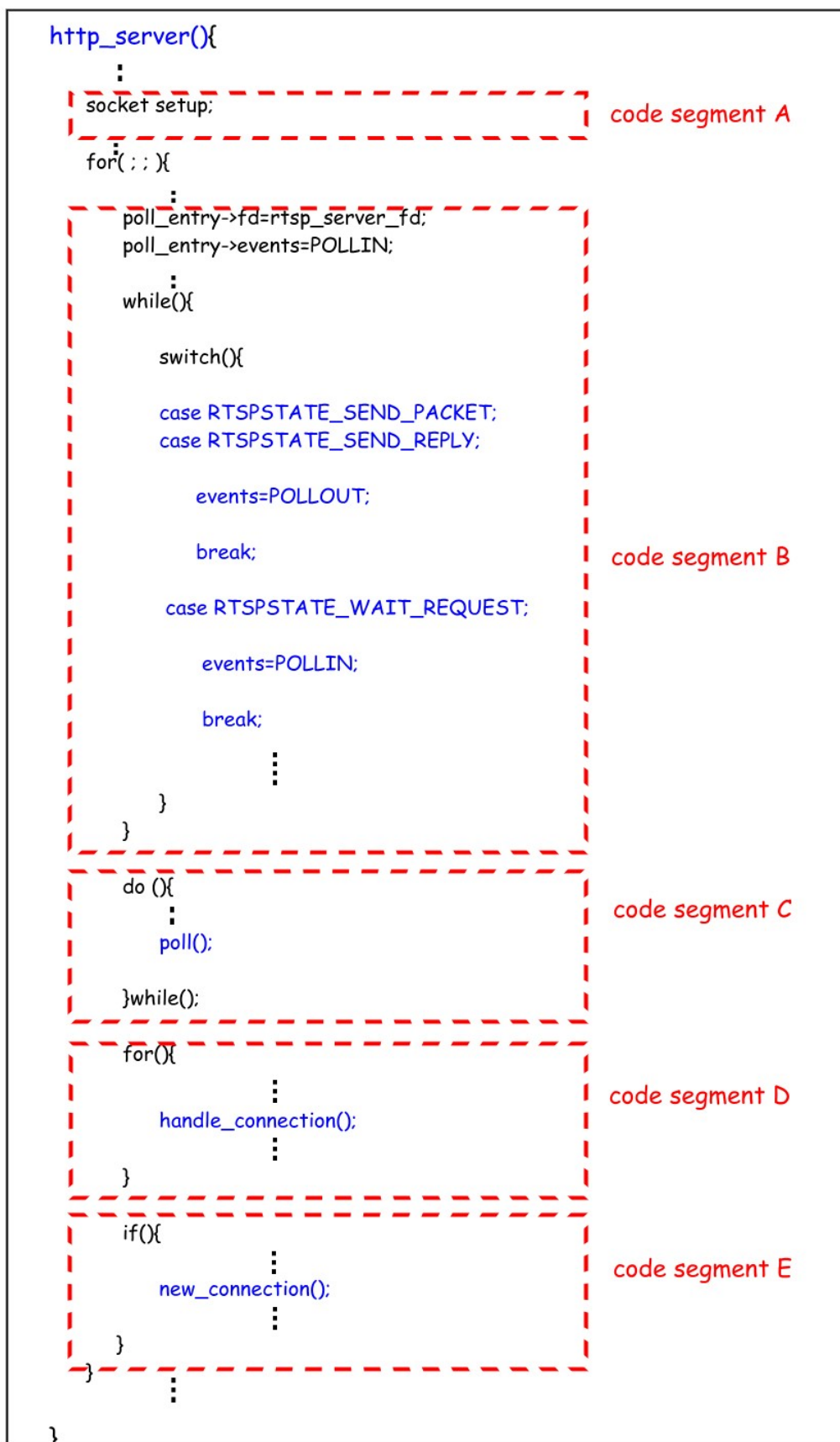
In this chapter, we present the overall system architecture and system flow chart of a streaming server.

### 3.1 System Architecture of A Streaming Server

Based on the functionalities a streaming server should have, we divide the system architecture into five segments in this work. And Figure 3-1 is the system architecture of a streaming server. Each segment consists of many actions and we would introduce each segment in the following sections.



**Figure 3-1 System Architecture of Streaming Server**



**Figure 3- 2 Pseudo Codes of http\_server()**

We discuss the single process streaming server. A single process means that only one process or thread is created and the system flow is sequential, and no context switch happens. Single process servers need to have an infinity loop which makes sure the server would always operate. And in Figure 3-2, we would see that there is an infinity loop indeed. Figure 3-2 is the pseudo codes of `http_server()`.

A streaming server has to check if there are new connections or new requests and to send the data to the clients. Firstly, a streaming server labels each connection a state which the connection is going to look for. The state may be a ready-to-send-packet state or a wait-for-service state or a ready-to-send-reply state and so on. Secondly, according to the state just labeled, server gives each connection different service, like parsing requests or sending packets. After the server completes the service of one client, the server goes on serving the other clients which are accepted and wait in the waiting list. If no other connections wait in the waiting list, the server then would check if there are new connections or new requests. And once a new connection is accepted, the server would label the connection state and go to serve the connection. The server repeats the four stages, state-labeling, service-giving, connection-checking, and

connection-accepting, until the server shuts down.

In Figure 3-2, we divide the overall system architecture into five code segments. The five code segments cooperate to achieve the three stages talked in the previous paragraph. For example, code segment B and code segment C are in charge of labeling states and checking connections, code segment D accounts for sending packets to the clients and parsing the requests from the clients, and code segment E is responsible for accepting new connections. Next we would introduce the five code segments which are under `http_server()`.

The followings are the five code segments in Figure 3-2. And we refer `SOCKET SETUP` to code segment A, `LABELING` to code segment B, `SELECT` to code segment C, `HANDLING` to code segment D, and `ACCEPT` to code segment E.

### **3.1.1 SOCKET SETUP ( Code Segment A ) In A Streaming Server**

Socket setup is the first step for every network programming. Hence, socket setup is also the first part in the streaming server's architecture. In Figure3-3, we show the detail components of the first part, SOCKET SETUP. There are a few important components in the SOCKET SETUP. They are :

- **socket ()**
  - The socket function is to create a socket. And the returned value of `socket()` is a socket descriptor which is very similar to a file descriptor. Just like we use `open()` to create a file descriptor and access the files in the disk. We use `socket()` to create a socket ,and we call this socket a listening socket.
  
- **bind()**
  - The bind function is to associate a socket with a network address.
  
- **connect()**
  - The connect function is to connect a socket to a remote network address.



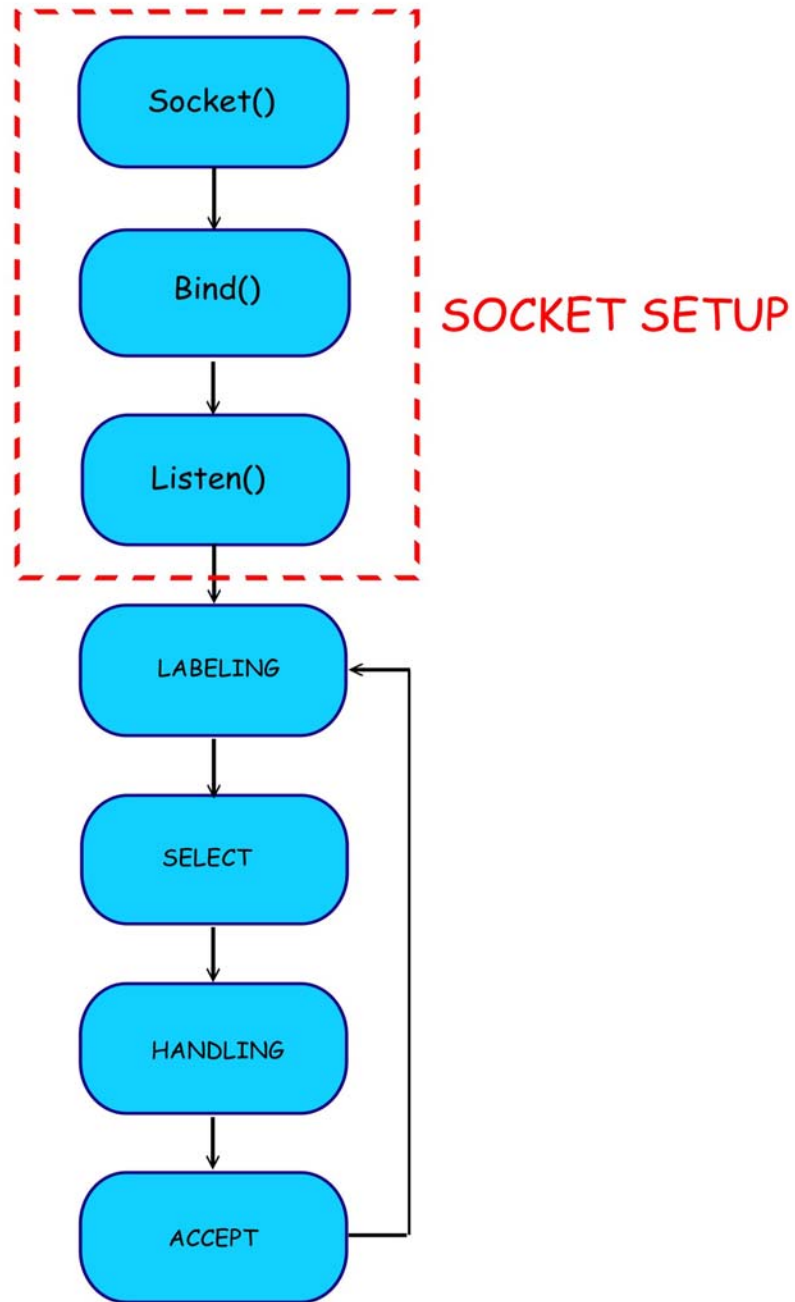
➤ `listen()`

- The `listen` function is to wait for incoming connection attempts.

➤ `accept()`

- The `accept` function is to accept incoming connection attempts. After the `accept` function is returned, a socket descriptor is given. This socket descriptor is called a connected socket descriptor and not a listening socket descriptor any more. Although `accept` is not categorized in `SOCKET SETUP` in this work, it is still a very important socket API in network programming.

Because the streaming server is a single process and only one listening socket is needed. Hence this code segment is not included in the infinity for loop. In this code segment, we only do `socket()`, `bind()`, and `listen()`, `accept()` is implemented in code segment E. Since code segment E is mainly for accepting new connections. Therefore, we do not implement `accept()` in this code segment. If the listening socket is successfully created, the process would go to the next code segment, code segment B.



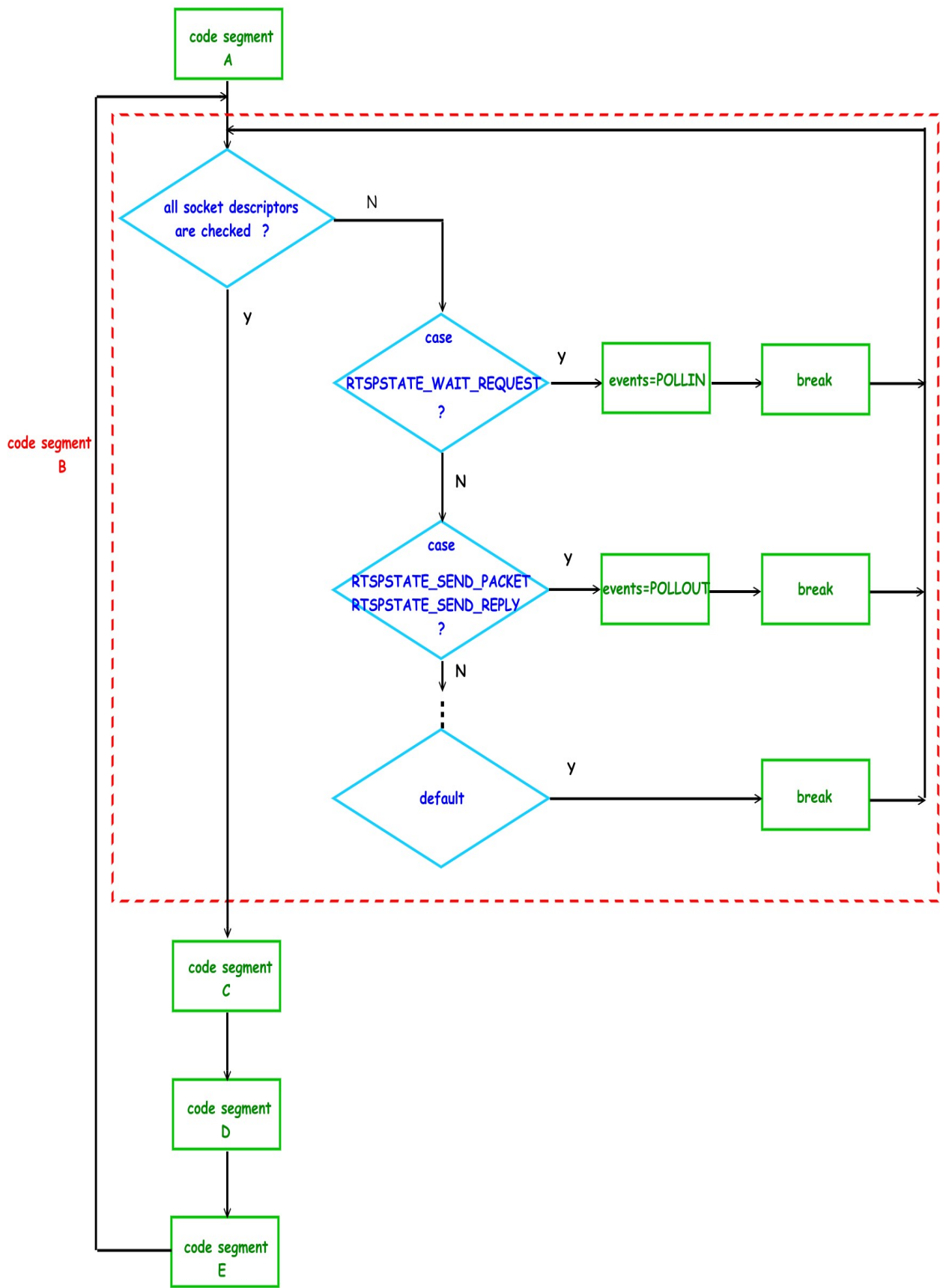
**Figure 3- 3 Components of SOCKET SETUP**

### **3.1.2 LABELING ( Code Segment B ) In A Streaming Server**

Because we have TCP sockets and UDP sockets in the streaming server, server has to give different sockets different events. And TCP sockets may transit its state between read or write state, we have to implement a state machine so that server can handle different states of TCP sockets.

In Figure 3-4, we show the flowchart of code segment B in more details. We could see that the streaming server would check all socket descriptors and assign each socket descriptor an events. But the listening socket descriptor is always assigned a POLLIN events in this code segment. Moreover listening socket descriptor does not have state, FFserver would go to the default case, do nothing, leave the switch multi-selection structure and go to check another socket descriptors. The variable events means that the action the socket descriptors is going to look for. For example, if a connected client is labeled a RTSPSTATE\_WAIT\_REQUEST state, then the connected client is going to look for a POLLIN event. And if a connected client is labeled a RTSPSTATE\_SEND\_PACKET state or a RTSPSTATE\_SEND\_REPLY state, then the connected

client will take care of a POLLOUT event. In any server, a connected client is a connection request by a client and the connection is accepted by the server. Once a connection is accepted, then the server would give this connection of the client a connected socket descriptor. That is, from then on, the connected socket descriptor represents the connection of the client. So the server could send or read data from the client by using the corresponding connected socket descriptor; that is, to send data to or read data from the connected socket descriptors is equivalent to send data to or read data from the connected clients. A more detail information about socket descriptors are discussed in section 3.3.1. A POLLIN event indicates that the connected socket descriptor is ready to read data from the connected client; a POLLOUT event means that the connected socket descriptor is going to send data from server to the client.

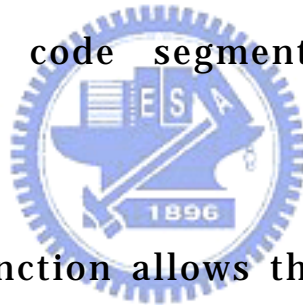


**Figure 3- 4 Flowchart of Code Segment B**

### 3.1.3 SELECT ( Code Segment C ) In A Streaming Server

Once each connected socket descriptor is given a events, we then need to check if the connected socket descriptor has responses and which state the connected socket descriptor belongs to.

This code segment is mainly to use select to notify kernel that there are new connections. Before we go on the discussion of this code segment, select should be introduced firstly.



The select function allows the process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. As an example, we can call select and tell the kernel to return only when


- any of the socket descriptors {3,9} are ready for reading, or
- any of the socket descriptors {4,6} are ready for writing, or
- any of the socket descriptors {5,7} have an exception

condition pending, or

- after 10 seconds have elapsed, where 3, 9, 4, 6, 5, and 7 in this work are socket descriptors.

That is, FFserver tells the kernel what socket descriptors FFserver is interested in (for reading, writing, or an exception condition) and how long to wait. Besides socket descriptors, any descriptor can be tested using select.

The select function is shown in the following block.



```
int select (int maxfd1, fd_set *readset, fd_set *writerset,  
           fd_set *exceptset, struct timeval *timeout) ;
```

There are five arguments maxfd1, readset, writerset, exceptset, and timeout in the select. The functions of these five arguments are explained as follow:

- maxfd1
  - This argument specifies the number of socket descriptors to be tested.

- readset
  - This argument defines the socket descriptors that we want the kernel to test for reading condition.
  
- writeset
  - This argument designates the socket descriptors that we want the kernel to test for writing condition.
  
- exceptset
  - This argument mentions the socket descriptors that we want the kernel to test for exception condition.
  
- timeout
  - This argument tells the kernel how long to wait for one of the specified socket descriptors to become ready.



Now a design problem is how to specify one or more descriptor values for each of these three arguments, readset, writeset, and exceptset. Select uses descriptor sets, typically an array of integers, with each bit in each integer corresponding to a socket descriptor. For example, using 32-bit integers, the first element of the array corresponds to socket descriptors 0 through 31, the second element of the array corresponds to descriptor 32 through 63, and so on. All the implementation details are



irrelevant to the application and are hidden in the `fd_set` datatype and the following four macros shown in Figure 3-5:

```
void FD_ZERO(fd_set *fdset);      /*clear all bits in fdset*/
void FD_SET(int fd, fd_set *fd_set); /*turn on the bit for fd in fdset*/
void FD_CLR(int fd, fd_set *fdset); /*turn off the bit for fd in fdset*/
int  FD_ISSET(int fd, fd_set *fdset); /*is the bit for fd on in fdset?*/
```

**Figure 3- 5 Four macros of `fd_set` datatype**

We allocate a socket descriptor set of the `fd_set` datatype, we set and test the bits in the set using these macros. For example, to define a variable of type `fd_set` and then turn on the bits for socket descriptors 3 and 9, we write the sample codes presented in Figure 3-6.

```
fd_set rset;

FD_ZERO(&rset); /*initialize the set: all bits off*/
FD_SET(3,&rset); /*turn on bit for fd 3*/
FD_SET(6,&rset); /*turn on bit for fd 6*/
```

**Figure 3- 6 Example of how `fd_set` is constructed**

Now we know more about `select` and `fd_set`, we could go on our work. In this segment, we set the socket descriptors into the corresponding `fd_set`. In Figure 3-7, we show the pseudo codes of code segment C and the flowchart of code segment C which is displayed in Figure 3-8. After entering this code segment, server firstly examines the events of the connected socket descriptors, then sets the connected socket descriptors into one of the `fd_sets`. For example, if the events of the connected socket descriptor is `POLLIN`, the discriminant “if (`fds[i].events & POLLIN`)” would be true, then server turns on the connected socket descriptor in the `read_set`. Otherwise, if the events of the connected socket descriptor is `POLLOUT`, server would turn on the connected socket descriptor in the `write_set`. After the connected socket descriptors are set in any one of the `fd_sets`, server calls `select` to help inspect whether the connected socket descriptors are ready to read or write. Once the `select` function is returned, the server examines if the discriminant “`FD_ISSET(fds[i].events,&read_set)`” is true, if it is true, server assigns the revents of the connected socket descriptor `POLLIN`. But if the discriminant “`FD_ISSET(fds[i].events,&write_set)`” is true, server appoints the revents of the connected socket descriptor `POLLOUT`. `select` is returned in this system only when new connections or RTSP methods that include

DESCRIBE, OPTIONS, SETUP, PLAY, PAUSE, and TEARDOWN appear. FD\_ISSET here copes with the situation that which socket descriptors are caught by select. This can help server to decide whom the data should be send to or receive from. The revents is used to show what events occurred and is an important variable in code segment E, so we discussed this variable in code segment E. After assigning the corresponding revents to the connected socket descriptors, the next step is to enter code segment D.



```

poll(){
    :
    fd_set read_set;
    fd_set write_set;
    fd_set exception_set;
    :
    FD_ZERO(&read_set);
    FD_ZERO(&write_set);
    FD_ZERO(&exception_set);
    :
    if (fds[i].events & POLLIN)
        FD_SET(fds[i].fd, &read_set);

    if (fds[i].events & POLLOUT)
        FD_SET(fds[i].fd, &write_set);

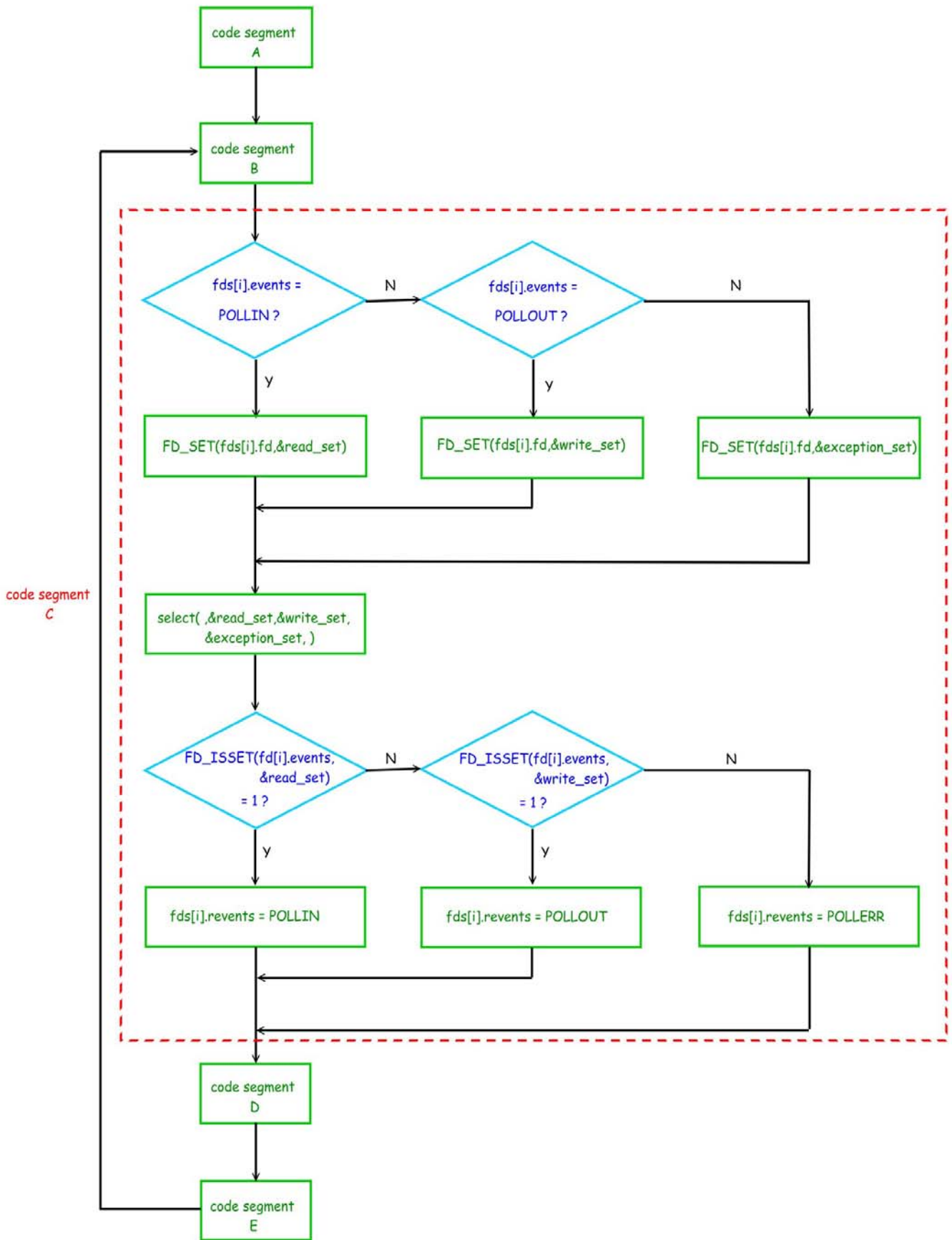
    if (fds[i].events & POLLERR)
        FD_SET(fds[i].fd, &exception_set);
    :
    select();
    if(FD_ISSET(fds[i].fd, &read_set))
        fds[i].revents=POLLIN;

    if(FD_ISSET(fds[i].fd, &write_set))
        fds[i].revents=POLLOUT;

    if(FD_ISSET(fds[i].fd, &exception_set))
        fds[i].revents=POLLERR;
    :
}

```

**Figure 3- 7 Pseudo Codes of Code Segment C**



**Figure 3- 8 Flowchart of Code Segment C**

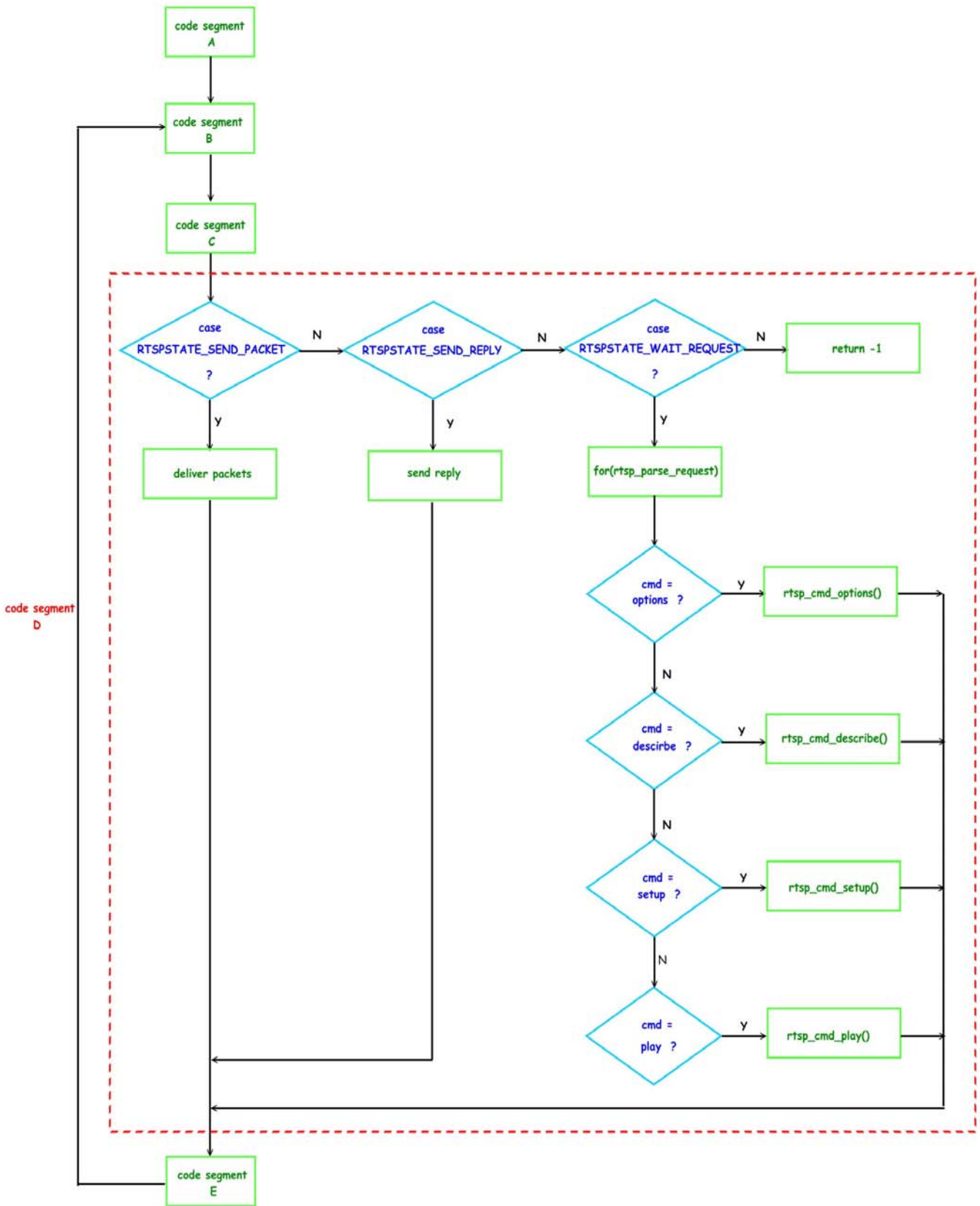
### 3.1.4 HANDLING ( Code Segment D ) In FFserver

After the connected socket descriptor is ready to read or write, then server starts to process the connected socket descriptor's requests.

Figure 3-9 is the flowchart of code segment D and Figure 3-10 is the pseudo code of this segment. Code segment D is responsible for the handlings of the requests of the connected socket descriptors, including parsing and sending packets. Server firstly checks the events of the connected socket descriptors. If the events of the connected socket descriptor is `RTSPSTATE_SEND_PACKET`, the server enters the action, `packets_deliver`. `Packets_deliver` consists of many sub-functions which mainly includes `packets_sending` and `timing_calculating`. `Packets_sending` is the action that sends the packets to the connected socket descriptor. `Timing_calculating` is mainly to calculate the PTS and DTS of the packets and the sending time of each packet. But the `packets_deliver` is not going to talk about in this work. Back to the switch case choices of the connected socket descriptors, if the events of the connected socket descriptor is `RTSPSTATE_WAIT_REQUEST`, the server would call the sub-function, `rtsp_parse_request()`, shown in Figure 3-11 . In function `rtsp_parse_request()`, server

parses the methods the connected socket descriptor requests and does the corresponding functions. For example, if the method the connected socket descriptor requests is OPTIONS, the server would call the function , `rtsp_cmd_options()` and does the codes in the function to fulfill the request of the connected socket descriptor. Code segment D is implemented with a for loop. The loop will repeat several times which are the number of the connected socket descriptors. In each loop of code segment D, if server is in a `packets_deliver` action, server would delivery just one frame of the file the connected socket descriptor requests and then finish this loop and go to another loop if there are more than two connected socket descriptors exist. And if server is in the sub-function `rtsp_parse_request()`, server would parse the RTSP method the connected socket descriptor requests and leave this sub-function when no further method is requested from the connected socket descriptor. In Figure 3-10, we can see that there is a `recv()` in case `RTSPSTATE_WAIT_REQUEST`. Server parses the connected socket descriptor's requests which are buffered in the buffer specified in the second argument in `recv()`. If there is a new RTSP method arrives from the connected socket descriptor, this method would be caught by `select` in code segment C. Then server uses `recv()` to receive this method, parses this method, and executes the

corresponding sub-function.



**Figure 3- 9 Flowchart of Code Segment D**



```

handle_connection(){
    switch(){
        ⋮
    case RTSPSTATE_WAIT_REQUEST:
        recv(c->fd,c->buffer_ptr,1,0);
        ⋮
        rtsp_parse_request();
        ⋮
        break;

    case RTSPSTATE_SEND_PACKET:
        ⋮
        send_data();
        ⋮
        break;
        ⋮
    case RTSPSTATE_SEND_REPLY:
        deliver_reply;
        c->state = RTSPSTATE_WAIT_REQUEST;
    }
}

```

**Figure 3- 10 Pseudo Codes of Code Segment D**

```

rtsp_parse_request(){
    extract command from url
    if(!strcmp(cmd,"DESCRIBE")){
        rtsp_cmd_describe(c,url);
    }
    else if(!strcmp(cmd,"OPTIONS")){
        rtsp_cmd_options(c,url);
    }
    ⋮
    other four methods
    ⋮
    c->state = RTSPSTATE_SEND_REPLY;
}

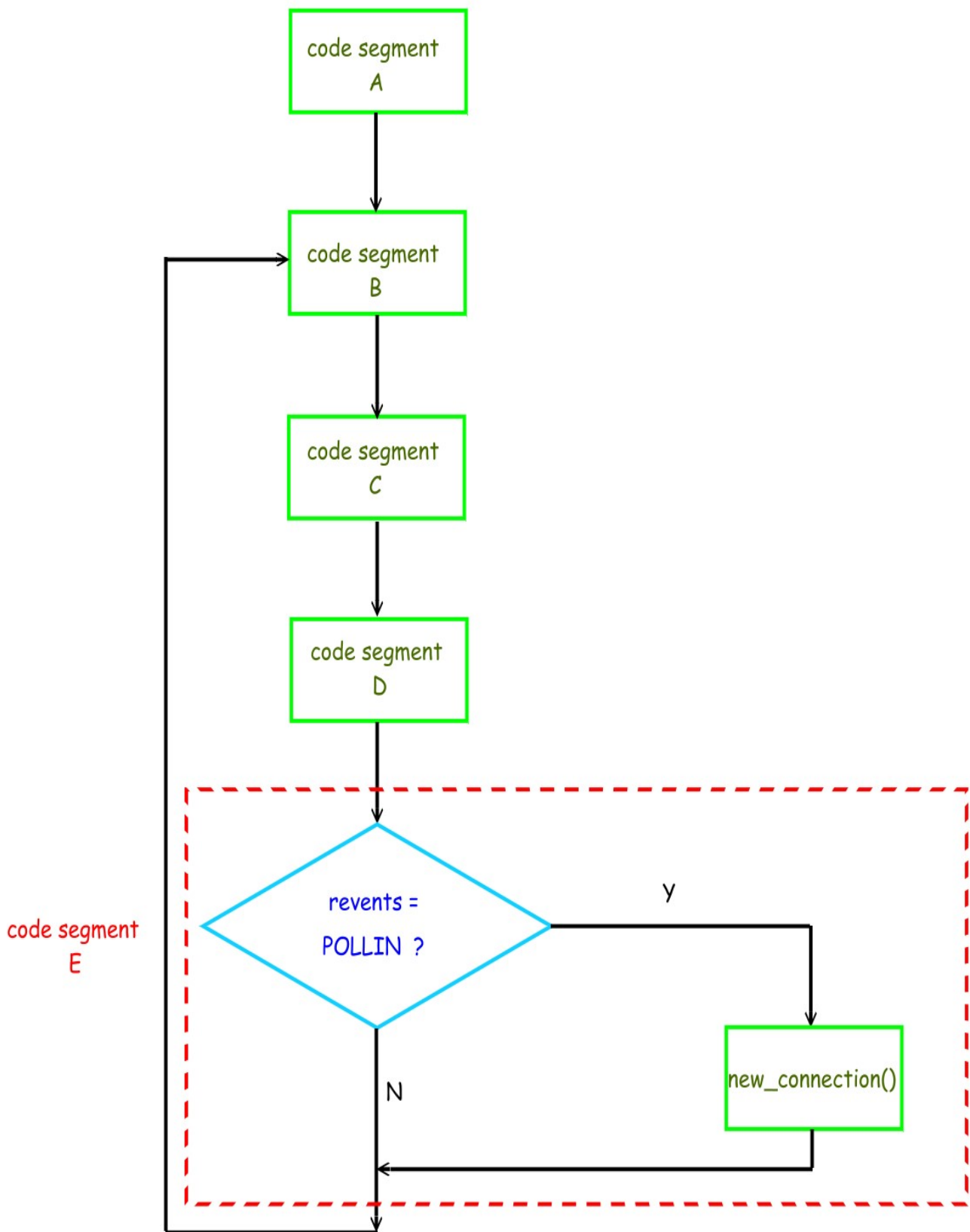
```

**Figure 3- 11 Pseudo Codes of rtps\_parse\_request()**

In Figure 3-11, there are many RTSP methods string comparisons. If the method the connected socket descriptor requests is DESCRIBE, the following function needs to be executed is `rtsp_cmd_descirbe()`. So do the other methods.

### **3.1.5 ACCEPT ( Code Segment E ) In FFserver**

Figure 3-12 is the flowchart of code segment E. In this code segment, server decides whether to accept a new connection. Server does not check the already connected socket descriptors but just checks the listening socket descriptor which is put in the `read_set`. If a new connection connects to server, then `select` would know the new connection in the `read_set`. And a variable `revents` is assigned to `POLLIN`. Variable `revents` indicates that the event which just occurred. Using this variable `revents` which is now `POLLIN`, server would enter the sub-function, `new_connection()`, accept this new connection, and give this new connection a new connected socket descriptor. After code segment E completes the work, server goes back to code segment B.



**Figure 3- 12 Flowchart of Code Segment E**

### 3.1.6 Overall Procedures In FFserver

After the five code segments are introduced in the above sections, now let's go through the overall procedures from code segment A to code segment E. We start server by executing the executable file `ffserver.c`. The process would be in the main function of `ffserver.c`. But the main function of `ffserver.c` just does some initializations. Therefore, we check the most important sub-function, `http_server()`. In `http_server()`, the first thing to do is to create a listening socket and this part is shown in code segment A. After the listening socket is initialized, FFserver enters the infinity loop. The first code segment in the infinity loop is code segment B. We make the events of the listening socket descriptor always POLLIN, as shown in Figure 3-2. Then at the beginning because no connection is accepted by FFserver, FFserver would pick the default case in the switch multiple-selection structure, leave this switch structure, and enter code segment C. In code segment C, we encounter a switch multiple-selection structure. For the reason that the events of the listening socket descriptor is POLLIN, so FFserver puts this listening socket descriptor in the `read_set` and calls `select`. Fortunately, a new connection from a client connects to FFserver, then `select` would return. `select` returns because the listening socket

descriptor in the `read_set` is set. Next, the revents of the listening socket descriptor is assigned `POLLIN`. Subsequently `FFserver` advances to code segment D. Although a new connection is caught by `select`, and `FFserver` moves to code segment D for handling the new connections' request. But this new connection has not been accepted by `FFserver` and given a new connected socket descriptor. So `FFserver` does nothing and goes to code segment E. This time `FFserver` would give this new connection a connected socket descriptor by calling `accept`. Now the new connection has its own connected socket descriptor and `FFserver` labels this connected socket descriptor `RTSPSTATE_WAIT_REQUEST`.

Now we go through this procedure again , and call this a second loop. This time the connected socket descriptor has been existed and has a `RTSPSTATE_WAIT_REQUEST` state. So in code segment B, the events of this connected socket descriptor is `POLLIN`. And in code segment C, this connected socket descriptor is put into the `read_set` since it has a `POLLIN` events. Subsequently, owing to the connected socket descriptor has requests, `select` would return. The revents of this connected socket descriptor is assigned `POLLIN`. Then `FFserver` enters code segment D. In code segment D, `FFserver` parses the requests from the connected socket

descriptor and executes the corresponding sub-function, like `rtsp_cmd_options` or `rtsp_cmd_describe`. When the request of the connected socket descriptor is `PLAY`, the state of this connected socket descriptor would be changed to `RTSPSTATE_SEND_PACKET` in sub-function `rtsp_cmd_play`. Subsequently, `FFserver` advances to code segment E and goes back to code segment B immediately since no new connection is caught by `select`.

Third loop begins with only one connected socket descriptor which has been done RTSP signaling. The procedure goes on as the second loop does, but action differs in code segment D. Now the state of the UDP socket descriptor of the connected socket is `RTSPSTAE_SEND_PACKET` because the state has been changed in `rtsp_cmd_play ()`, `FFserver` would enter `packets_deliver` and start to send one frame of the file the connected socket descriptor requested in the RTSP signaling. After `FFserver` sends one frame according to the DTS and PTS, `FFserver` leaves this segment and goes to code segment E. Until now, we complete third loop.

If there is no new connection, in the fourth loop `FFserver` would do the same things as those do in third loop. If there is a new connection, then in the fourth loop `FFserver` would serve the old connected socket descriptor,

catch this new connection by calling `select`, accept this new connection by calling `accept`, and give this new connection a different connected socket descriptor. Then in the later loops, `FFserver` would take turns serving this two connected socket descriptors.

## **3.2 RTSP Methods Implemented In `FFserver`**

We now examine the pseudo codes of the four RTSP methods which are implemented in `FFserver`.

### **3.2.1 `RTSP_CMD_OPTIONS()`**



The beginning method requested by VLC is `OPTIONS`, so we check the pseudo code of `rtsp_cmd_options()` firstly. Figure 3-13 exhibits the pseudo codes, and this function just edits the RTSP `OPTIONS` replies. And a detail description about `OPTIONS` is depicted in section 2.1.3 .

```

rtsp_cmd_options(){
    ⋮
    url_fprintf(c->pb,"RTSP/1.0%d%s\r\n",RTSP_STATUS_OK,"OK");

    url_fprintf(c->pb,"CSeq:%d\r\n",c->seq);

    url_fprintf(c->pb,"Public:%s\r\n","OPTIONS,DESCRIBE,SETUP,TEARDOWN,PLAY,PAUSE");
    ⋮
}

```

**Figure 3-13 Pseudo Codes of rtsp\_cmd\_options()**

### **3.2.2 RTSP\_CMD\_DESCRIBE()**

After the OPTIONS method is done, the second method is DESCRIBE. Figure 3-14 is the pseudo codes of rtsp\_cmd\_describe(). According RFC 2326, DESCRIBE reply usually includes sdp, and we certainly see a function, called prepare\_sdp\_description() which is responsible for sdp preparations. Figure 3-15 is the pseudo codes of the function, prepare\_sdp\_description(). Because DESCRIBE reply should includes the codec type of the file the connected socket descriptor requests, there should be a part called parsing. In Figure 3-15, there is a sub-function rtp\_get\_payload\_type() which lets FFserver decide the payload type of the file by checking the array listed in Table 3-1.



```

rtsp_cmd_describe(){
    ⋮
    prepare_sdp_description();

    url_fprintf(c->pb,"Content-Base:%s\r\n",url);

    url_fprintf(c->pb,"Content-Type:application/sdp\r\n");

    url_fprintf(c->pb,"Content-Length:%d\r\n",content_length);

    ⋮
}

```

**Figure 3- 14 Pseudo Codes of rtsp\_cmd\_describe()**



```

prepare_sdp_description(){
    ⋮
    rtp_get_payload_type();

    url_fprintf(pb,"v=0\n");

    url_fprintf(pb,"o=- 0 0 IN IP4 %s\n",ipstr);

    url_fprintf(pb,"s=%s\n",title);

    url_fprintf(pb,"m=%s%d RTP/AVP %d\n",mediatype,port,payload_type);

    ⋮
    url_fprintf(pb,"a=control:streamid=%d\n",(i+1)%2);

    ⋮
}

```

**Figure 3- 15 Pseudo Codes of prepare\_sdp\_description()**

pt	encoding name	audio(a) video(v)	clock rate (Hz)	channels (audio)
0	PCMU	A	8000	1
1	reserved			
2	reserved			
3	GSM	A	8000	1
4	G723	A	8000	1
5	DVI4	A	8000	1
6	DVI4	A	16000	1
7	LPC	A	8000	1
8	PCMA	A	8000	1
9	G722	A	8000	1
10	L16	A	44100	2
11	L16	A	44100	1
12	QCELP	A	8000	1
13	CN	A	8000	1
14	MPA	A	90000	
15	G728	A	8000	1
16	DVI4	A	11025	1
17	DVI4	A	22050	1
18	G729	A	8000	1
19	reserved	A		
20-23	unassigned	A		
24	unassigned	V		
25	CeLB	V	90000	
26	JPEG	V	90000	
27	unassigned	V		
28	nv	V	90000	
29-30	unassigned	V		
31	H261	V	90000	
32	MPV	V	90000	
33	MP2T	AV	90000	
34	H263	V	90000	
35-71	unassigned	?		
72-76	reserved for RTCP conflict avoidance			
77-95	unassigned	?		
96-127	dynamic	?		

**Table 3- 1 Payload Types Constructed In FFserver**

### 3.2.3 RTSP\_CMD\_SETUP()

Clients receive the DESCRIBE reply and then send the SETUP method. So we talk about the pseudo codes of `rtsp_cmd_setup()` which is shown in Figure 3-16.

```
rtsp_cmd_setup(){
    :
    rtp_new_connection();
    :
    open_input_stream();
    :
    rtp_new_av_stream();
    :
    url_fprintf(c->pb,"Session:%s\r\n",rtp_c->session_id);

    url_fprintf(c->pb,"Transport:RTP/AVP/UDP;unicast;client_port=%d-%d;server_port=%d-%d",
        th->client_port_min,th->client_port_min+1,port,port+1);
    :
}
}
```

**Figure 3- 16 Pseudo Codes of `rtsp_cmd_setup()`**

In Figure 3-16, there are three sub-functions, `rtp_new_connection()`, `open_input_stream()` and `rtp_new_av_stream()`. The followings are the introductions to three sub-functions.

- `rtp_new_connection()`
  - This sub-function does some SETUP method messages initializations and is not as important as the other two sub-functions.
  
- `open_input_stream()`
  - From the name of the sub-function, we know it is about the opening of the file stream. This sub-function also tells the client from where the file should play.
  
- `rtp_new_av_stream()`

This sub-function accounts for the setup of the file stream which is just opened by `open_input_stream()`.

### 3.2.4 RTSP\_CMD\_PLAY()

The last method, PLAY method, shows its pseudo codes in Figure 3-17. In Figure 3-17, a subfunction, `find_rtp_session_with_url()`, sends the streams into their sessions according to the mechanism specified in SETUP. As we mentioned in the above discussion, the state of the connection would change from `RTSPSTATE_WAIT_REQUEST` to `RTSPSTATE_SEND_PACKET` when FFserver gets a PLAY request from the client. FFserver would create a UDP socket which is responsible for sending RTP packets to the connected socket descriptor. And this UDP socket descriptor is assigned a `RTSPSTATE_SEND_PACKET` state and a `POLLOUT` events. The connected socket descriptor remains a `RTSPSTATE_WAIT_REQUEST` state.

```
rtsp_cmd_play(){
    ⋮
    find_rtp_session_with_url();
    ⋮
    url_fprintf(c->pb,"Session:%s\r\n",rtp_c->session_id);

    rtp_c->state = RTSPSTATE_SEND_PACKET;
    ⋮
}
```

**Figure 3-17 Pseudo Codes of `rtsp_cmd_play()`**

If the state of the connected socket descriptor is `RTSP_STATE_SEND_PACKET`, when to send the packets is an important issue. Here, we introduce two timestamps, presentation timestamp ( PTS ), and decode timestamp ( DTS ). We explain these two as follows:

➤ PTS

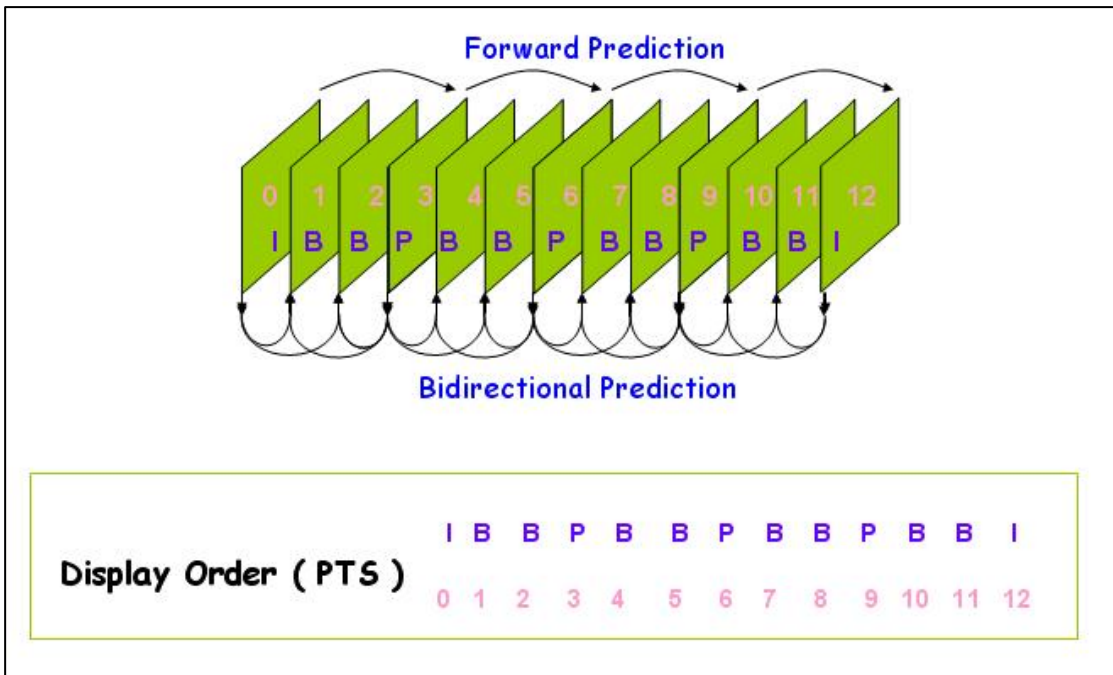
- The timestamp indicates that when the picture must be presented or displayed.

➤ DTS

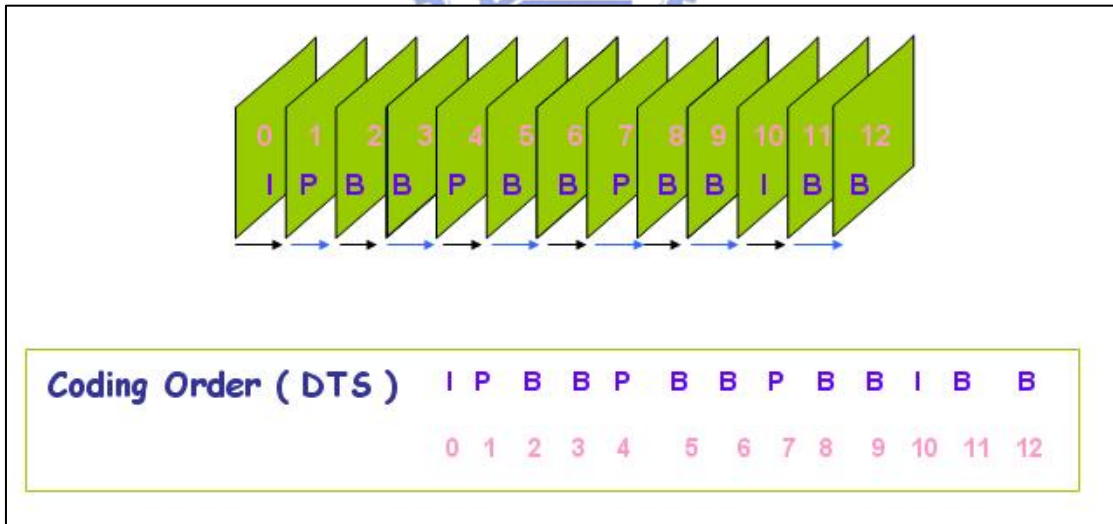
- The timestamp demonstrates when the picture must be decoded.



After constructing these two timestamps, FFserver can easily decide when to send the packets. We show PTS and DTS in Figure 3-18 and Figure 3-19 respectively.



**Figure 3- 18 PTS : the pictures' display order**



**Figure 3- 19 DTS : the pictures' coding order**

# CHAPTER 4

## EXPERIMENTAL RESULTS

In this chapter, we show some experimental results which including pictures before and after the modifications of RTSP in FFserver and timing of each code segment.

### 4.1 Results before and after the modifications in FFserver

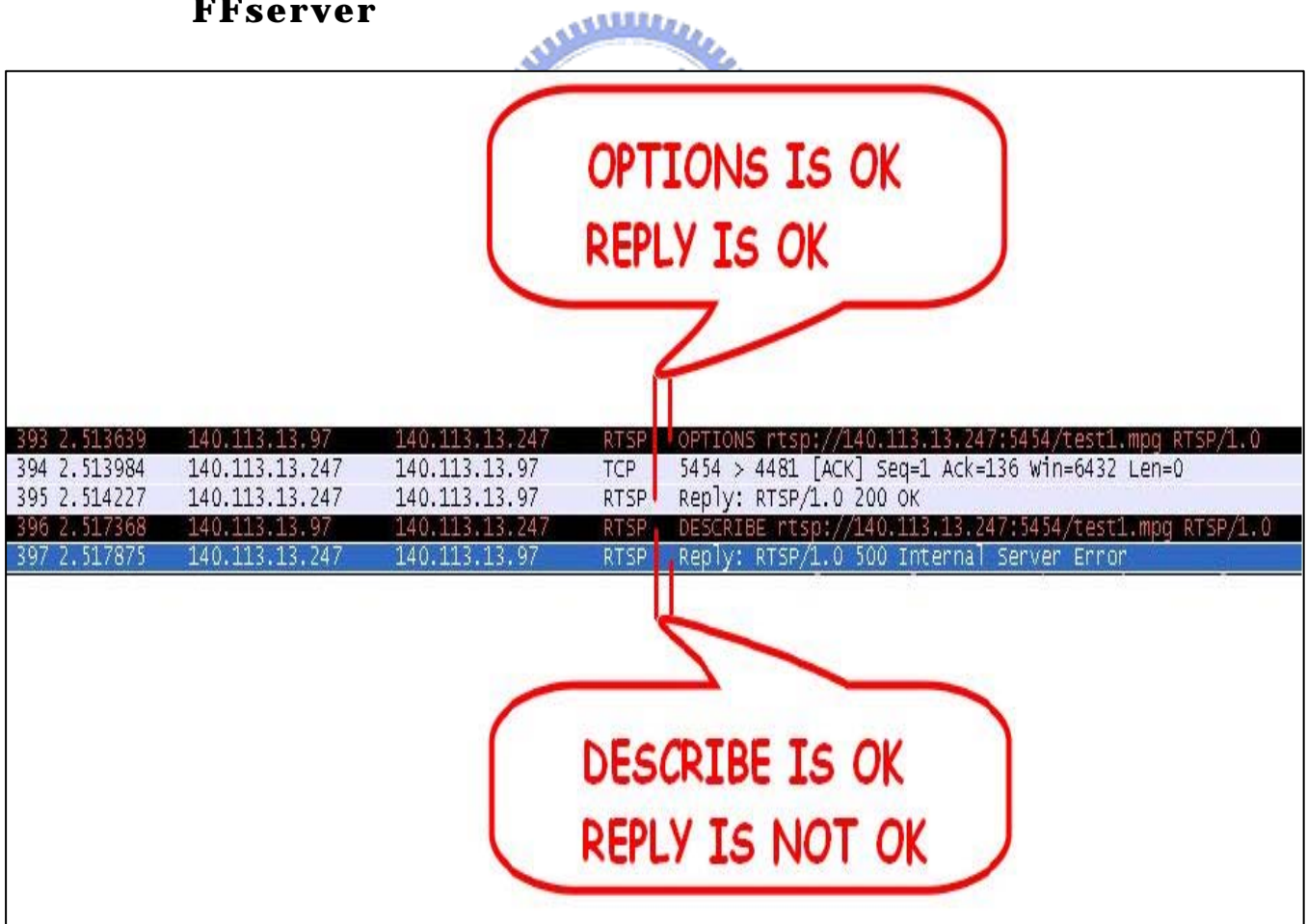


Figure 4-1 Old ffserver's packets captured by Ethereal



Figure 4-1 is the packet of old FFserver captured by Wireshark. We can see that there is an error happened in the reply of the DESCRIBE method. After we did the first two modifications specified in section 2.3, the result is shown in Figure 4-2.

291	1.845971	140.113.13.97	140.113.13.247	RTSP	OPTIONS rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
292	1.846340	140.113.13.247	140.113.13.97	TCP	5454 > 1446 [ACK] Seq=1 Ack=136 Win=6432 Len=0
293	1.846583	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK
294	1.849025	140.113.13.97	140.113.13.247	RTSP	DESCRIBE rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
295	1.849792	140.113.13.247	140.113.13.97	RTSP/S	Reply: RTSP/1.0 200 OK, with session description
296	1.856860	140.113.13.97	140.113.13.247	RTSP	SETUP rtsp://140.113.13.247:5454/test1.mpg/streamid=0 RTSP/1.0
297	1.858261	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK
298	1.862390	140.113.13.97	140.113.13.247	RTSP	SETUP rtsp://140.113.13.247:5454/test1.mpg/streamid=1 RTSP/1.0
299	1.863366	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 461 Unsupported transport
300	1.864228	140.113.13.97	140.113.13.247	RTSP	PLAY rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
301	1.865309	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK

**SETUP FOR SECOND  
STREAM IS NOT OK**

**Figure 4- 2 Packets after first modifications**

Although we had done the first two modifications, there is still an error. In order to debug this error, we spent a lot of time on it. Finally, we did the third

modification discussed in section 2.3, and the result is displayed in Figure 4-3. Until now, VLC can connect to FFserver by RTSP.

1497	9.084378	140.113.13.97	140.113.13.247	RTSP	OPTIONS rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
1498	9.084730	140.113.13.247	140.113.13.97	TCP	5454 > 4327 [ACK] Seq=1 Ack=135 Win=6432 Len=0
1499	9.084976	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK
1500	9.088088	140.113.13.97	140.113.13.247	RTSP	DESCRIBE rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
1501	9.089109	140.113.13.247	140.113.13.97	RTSP/S	Reply: RTSP/1.0 200 OK, with session description
1505	9.106613	140.113.13.97	140.113.13.247	RTSP	SETUP rtsp://140.113.13.247:5454/test1.mpg/streamid=1 RTSP/
1506	9.108136	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK
1507	9.113563	140.113.13.97	140.113.13.247	RTSP	SETUP rtsp://140.113.13.247:5454/test1.mpg/streamid=0 RTSP/
1508	9.116590	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK
1509	9.118168	140.113.13.97	140.113.13.247	RTSP	PLAY rtsp://140.113.13.247:5454/test1.mpg RTSP/1.0
1512	9.120963	140.113.13.247	140.113.13.97	RTSP	Reply: RTSP/1.0 200 OK

AFTER SOME MODIFICATIONS  
ALL REPLIES ARE OK !!

**Figure 4- 3 All modifications are completed**

## 4.2 Procedure Of Code Segments

code segment	time elapsed	code segment	time elapsed
A	42	C	5
C	4194	D	1
C	749084	D	10
E	34	B	2
B	2	C	2443
C	5	D	2
D	18	D	207
B	2	B	2
C	3752	C	5
D	298	D	483
B	2	D	9
C	6	B	2
D	9	C	14481
B	2	D	2
C	18989	B	2
D	1023	C	15989
B	2	D	3

**Figure 4- 4 Procedure of code segments**

In Figure 4-4, the procedures of each code segment is shown. We can see that the procedure is the

same as the one we discussed in section 3.1.6. Code segment A is performed only one time. And if there is a connected socket descriptor, the procedure would go in sequence from code segment B to code segment D and repeat until the connected socket descriptor is down. And the detail time used by each code segment is shown in the appendix . In appendix , we can see that sending is to send a packet and the time in the bracket in the time elapsed is the sending time.



## **CHAPTER 5**

### **CONCLUSIONS AND FUTURE WORKS**

In this thesis, we present the system architecture and system flowchart of FFserver. We also show how to modify FFserver so that VLC could connect to FFserver by using RTSP signaling.

After the modifications are completed, VLC could connect to FFserver and play multimedia data which are streamed by FFserver. If the multimedia data are MPEG1 files, the quality of play is quite good on VLC. But if the multimedia data are MPEG2 or MPEG4 files, the quality would not as good as that of MPEG1 file on VLC.

In the future, we hope that FFserver would be modified very well so that all clients could play in a better quality no matter what kind of multimedia data streamed by FFserver.

The most important issue in any streaming server is the timing control. If the timing control mechanism is good, the streaming server is more stable.

## References

- [1] “FFmpeg” available at <http://ffmpeg.mplayerhq.hu/> .
- [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] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, ”Hypertext Transfer Protocol – HTTP/1.1”,
- [5] “Wikipedia” available at [http://en.wikipedia.org/wiki/Main\\_Page](http://en.wikipedia.org/wiki/Main_Page) .
- [6] “RTP Payload Type” available at <http://www.iana.org/assignments/rtp-parameters>
- [7] Chien-hua James Chen, ”Design and Implementation of Real-time Interactive RTP/RTSP Multimedia Streaming Monitoring System with Bandwidth Smoothing Technique,” Master thesis, The Department of Communication Engineering, National Chiao Tung University.
- [8] Jong-Shou Wu, ”Analysis of Streaming Server’s Properties,” Master thesis, The Department of Communication Engineering, National Chiao Tung University.
- [9] Wei-Tung Chang, “Development and Implementation of Multimedia Streaming Platform,” Master thesis, The Department of Communication Engineering, National Chiao Tung University.

## Appendix

Code Segment	Time Consumed (uSec)	
B	4	
C	15478	
D	12	
B	4	
C	15979	SENDING
D	65(10)	
B	4	
C	15931	SENDING
D	85(8)	
B	3	
C	15903	
D	17	
B	45	
C	15938	
D	20	
B	4	
C	15973	SENDING
D	45(8)	
B	4	
C	15948	
D	19	
B	4	
C	15973	SENDING
D	98(9)	
B	4	
C	15894	
D	19	
B	4	
C	15973	SENDING
D	62(8)	
B	47	
C	15886	
D	19	



B	4	
C	15972	SENDING
D	87(9)	
B	4	
C	15902	
D	12	
B	4	
C	16025	SENDING
D	90(12)	
B	4	
C	15861	
D	35	
B	5	
C	20224	SENDING
D	90(11)	
B	72	
C	15550	
D	22	
B	5	
C	15990	SENDING
D	79(9)	
B	4	
C	15889	
D	13	
B	4	
C	15975	SENDING
D	89(10)	
B	4	
C	15903	
D	19	
B	4	
C	15970	SENDING
D	80(7)	
B	48	
C	15869	



D	12	
B	4	
C	15978	SENDING
D	46(7)	
B	5	
C	15946	
D	13	
B	4	
C	15978	SENDING
D	103(17)	
B	4	
C	15887	
D	41	
B	4	
C	15951	
D	32	
B	42	
C	15919	SENDING
D	125(10)	
B	4	
C	15867	SENDING
D	37(6)	
B	3	
C	15975	
D	17	
B	4	
C	15954	SENDING
D	30(6)	
B	4	
C	15966	
D	25	
B	5	
C	15958	SENDING
D	101(11)	
B	5	
C	15826	

D	23	
B	4	
C	15966	
D	19	
B	4	
C	15989	SENDING
D	48(6)	
B	4	
C	15927	SENDING
D	39(7)	
B	4	
C	15950	
D	12	
B	4	
C	15981	SENDING
D	82(10)	
B	12	
C	15856	
D	12	
B	3	
C	15978	
D	26	
B	4	
C	15965	SENDING
D	104(6)	
B	4	
C	15910	SENDING
D	30(7)	
B	4	
C	15940	
D	13	
B	4	
C	15977	SENDING
D	130(9)	
B	4	
C	15862	

D	13
---	----

