國立交通大學

電機學院通訊與網路科技產業研發碩士班

碩 士 論 文

串流平台多使用者的傳輸時間分析

Multi-user Timing Control in Streaming Server

研 究 生：曹素仙

指導教授：張文鐘　教授

中 華 民 國 九 十 六 年 八 月

串流平台多使用者的傳輸時間分析

Multi-user Timing Control in Streaming Server

研 究 生：曹素仙　　Student：Su-Hsian Tsao

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學

電機學院通訊與網路科技產業研發碩士班

碩 士 論 文

A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Industrial Technology R & D Master Program on

Communication Engineering

August 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月

串流平台多使用者的傳輸時間分析

研究生：曹素仙　　指導教授：張文鐘 博士

國立交通大學電機學院產業研發碩士班

摘 要

串流伺服器因為必須同時服務多個用戶的連線需求以及要滿足即時傳送封包給不同用戶，因此如何設計一個順序來控制每張 frame 的傳送時間是本論文重點。

伺服器會對每一張 frame 計算其 PTS(display order)和傳送時間 DTS(Sending order)，並且依照 DTS(Sending order)來傳送該 frame。當某個使用者的 frame 所切割出來的 payload 的 DTS(Sending order)小於系統時間(Wall clock)時，意即該 payload 可以開始經過封裝傳送的處理，而在此需特別強調的是同一個 frame 所切割出來的 payload 都會具有同樣的 DTS(Sending order)。

本論本研究的核心所在便是去研究串流伺服器如何使用單一行程來達到多工，其時間排程的機制、以及設計理念。

Multi-user Timing Control in Streaming Server

Student : Susan Tsao Advisor : Dr. Wen – Thong Chang

Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

Abstract

Streaming servers need to consider the simultaneous transmission requirement for all users, thus how to design a mechanism to control every packet's transmission time will be a critical issue in the server design.

For real time transmission, server would calculate a sending order for each frame. This order is denoted as DTS. The packets associated with a frame are sent according to DTS of a frame. During transmission the DTS of a payload of one user is compared with the Wall clock and packets are sent sequentially.

The thesis is to discuss the method that can be used to maintain a smooth transmission order for all frames in a multi-user environment such that the real time transmission can be achieved and the network is not over-flown.

致 謝

能順利完成這篇論文，最感謝的人是我的指導教授 張文鐘 博士。 在這兩年的碩士生涯中，感謝老師在我的研究上的指導，使我能順利地完成學業。同時也謝謝 廖維國教授，林大衛教授以及何文楨主任於口試時的指導，有了您們的指導才使得這篇論文更趨完備。

另外要感謝實驗室裡的好伙伴們，包含瑩甄，孟潔，程翔，榮勝，新畢，書維以及政鴻，同時再次的感謝邱程翔在我寫論文時給我在一些觀念上的澄清。還有我們實驗室可愛的學弟，宗學，文賢，明山，峻權以及政達，實驗室裡有你們就會有歡樂聲，謝謝你們的陪伴。而特別要感謝吳宗修學長給於我一些時間讓我問問題，在此謝謝你。

最後我要感謝的是我最偉大的母親，在我無助時給予鼓勵。還有我們家裡的大哥，大嫂，二哥，小弟以及我的侄女小慈慈，謝謝你們陪伴，使我在心情煩悶時陪我渡過。因為有你們的陪伴以及支持讓我能順利地完成碩士學業，再次的謝謝大家。

致於 2007.8 風城 交大

Susan

目錄

圖目錄

表目錄

一、緒論

1.1 串流伺服器簡介

當我們在網頁上點選一個檔案的開啟時，一般所提供的檔案下載與播放的伺服器做法為：首先將伺服器開啟後，使用者端要求傳檔案，此時伺服器與使用者將會連線，連線成功，伺服器與使用者端就開始檔案的傳輸，直到檔案傳輸完畢，才會關閉此一連線。因為伺服器所傳到使用者端的資料內容有前後的相關性，所以必須要將整個檔案傳輸完畢後，使用者端才能處理這份檔案。以上的方法只適合資料量(幾 K Bytes 到幾 M Bytes 之間，如 JPEG 圖檔，Word 文件等)較少的傳輸方式，而且使用者等待時間大概幾秒到幾分鐘，這樣的傳輸時間對一個使用者來說還可以接受的範圍。

如果目前要傳輸的是一個多媒體檔案，檔案較大(如電影檔案，聲音檔案等)這樣的檔案資料量都很 大(幾 M Bytes 幾 G Bytes)，當使用者想看一段的電影或是聽一段的音樂時，在一般的伺服器上必須要等待整個檔案傳輸完畢才能處理，而且必須要花費數個鐘頭，將會很浪費時間。因此我們需要另一種傳輸的方式，而使用串流的傳輸方式則是檔案傳輸的另一個選擇。

如果我們要讓一個檔案符合串流的傳輸方式，則必須要減少檔案的資料內容間前後相關性的問題。使得一台伺服器傳輸多少資料，而使用者端就處理多少資

料，以減少使用者等待資料傳輸的時間。串流伺服器則根據使用者端的要求，將傳輸的檔案切割成一個個資料，使每份資料獨立，並以穩定的網路傳輸速率傳送到使用者端，讓使用者可以根據接收到的封包來處理資料，因此可以在很短的時間內，滿足使用者的需求。

接下來介紹的是串流傳輸的優點以及串流所用到的兩大協定分類~

串流傳輸的優點有下面三點:

1.即時播放 – 不需等待影片全部下載，就可以即時收看。

2.現場播放 – 在網路上現場直播節目是串流傳輸唯一的方法。

3.節省空間 – 資料在做串流傳輸時，不會在使用者的電腦上留下檔案，也不需擔心硬碟空間不足夠的問題。

而串流的協定主要分為兩大類，也就是 TCP(Transmission Control Protocol) 與 UDP(User Datagram Protocol)，如下圖 1.1 所示。原則上我們會使用 TCP 來傳送控制信令，UDP 來傳送多媒體影音封包，例如 YouTube 以及 Nuuno 監控系統的應用。YouTube 它所提供的服務為可上傳影片檔案，網路上的每位使用者便可以連上該網站做即時影片的觀賞; Nuuno 則是使用戶端可透過該系統同時監控由伺服端所傳送的影像即時封包，並且透過圖形使用者介面(GUI)，用戶可以監控多個不同地點攝影機所擷取的影像資料，歸納起來上述的應用都是基於 Client / Server 的架構。

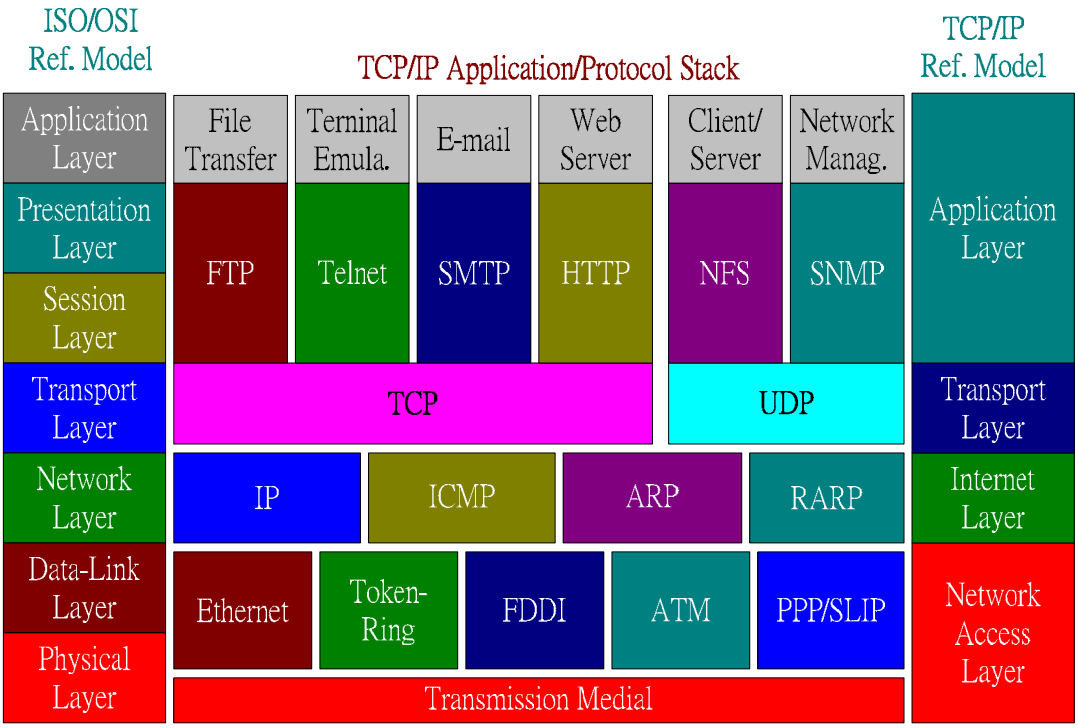


圖 1. 1　TCP/UDP Protocol

1.2　研究背景與動機

在多媒體串流伺服器中，行程的多工配置決定了伺服器效能的好壞，目前常見的多工型態有 multi-process、multi-thread 和 event-trigger 三種。我們可以發現在高效能的串流伺服器中，常見到的是以 multi-thread 和 event-trigger 為主的行程多工配置，因為這兩種佔用 CPU 的資源較少，故 CPU 在單位時間內將可以完成更多使用者所交付的任務，但相對的其程式設計的難度也較高。以串流伺服器中的開放原始碼 ffserver 與 live555 來說，其多工型態與信令溝通的協定便不盡相同。

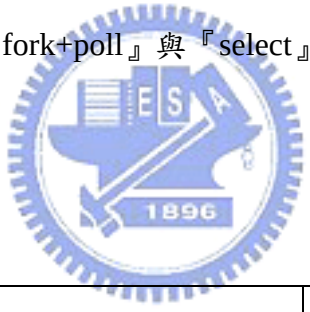
以 ffserver 來說，其整體設計以『簡單、易用』為主，因為在信令溝通以及封包傳送上，ffserve 均以 HTTP 協定來完成之，採用 HTTP 協定最大的優勢為封包可以穿越防火牆的阻擋，並且一般的使用者可以直接使用如微軟的 Internet Explorer 瀏覽器來接收來自伺服器的影音封包。

而 live555 其整體設計著重於『效率、功能』，『效率』指的是其多工是以 event-trigger 的單一行程來完成之，故在沒有用戶連線的狀況下，CPU 便不需要浪費資源去 polling Socket 有沒有封包進來；『功能』指的是 live555 依循 RFC 2326 RTSP 的標準來實作串流伺服器的溝通協定，RTSP 的最大優點便是提供了使用者可以要求伺服器對目前所串的影音進行快轉、迴轉以及暫停的動作。此外，一般使用者在使用上會覺得比 RTSP 伺服器略為複雜，因為使用者大多必須安裝特定的播放器(如 VLC player)才能觀賞到伺服器所傳送的聲音影像，其原因是各家的伺服器在 RTSP 信令的實作上可能不盡相同，導致彼此無法相容，例如我們 Lab 所開發研究的 Nctu_ImageLab(資策會版本)伺服器/用戶端便無法與 VLC player 互連，原因便是 RTSP 信令不相容，上述三種伺服器的重要特性如下表 1.1 所示。

編號	伺服器名稱	溝通信令協定	多工型態	備註
1	ffserver	HTTP + RTSP	select	Single-process (Event-trigger)
2	live555	RTSP	select	Single-process (Event-trigger)
3	Nctu_ImageLab	RTSP	thread+select	Multi-thread (Event-trigger)

表 1. 1　伺服器特性比較表

最後提到多工型態部分，下表二列出三種伺服器常用的多工型態，其中『thread+select』可以說是『fork+poll』與『select』的折衷型態，詳細比較可參考下表 1.2 的比較說明。



編號	多工型態	優點	缺點
1	fork+poll	1.程式撰寫/流程控制簡單	1. fork 在記憶體使用上用量較多，且記憶體回收問題需特別注意 2. IPC 問題 (inter-process communication) 3. 伺服器效率最差 4. 程式除錯最困難
2	select (Single-process)	1.伺服器效率較佳 2.程式除錯較簡單	程式撰寫/流程控制最複雜

3	thread+select (Multi-thread)	使用者可以依多工任務需求配置執行緒，多工任務劃分清楚	IPC 問題 (IPC -行程間的通訊) (Zombie Process 的問題)

表 1. 2　多工型態分析

由上面的解說我們可以理解到多工型態的優缺點，而實際上『設計面』往往取決於『應用面』，因此在程式設計時我們便要事先考量伺服器的應用為何，進而選擇恰當的多工型態 。 舉例來說，fork 會替每一個用戶連線複製同樣的一份程式碼，因此會消耗較多的記憶體，諸如此類的軟體問題也都會連帶牽涉到系統的硬體配置，以目前的嵌入式系統應用來說，軟硬體協同設計已是趨勢，所以多工型態的選擇確實是值得我們去研究的議題。

1.3　論文架構

瞭解了伺服器的應用後，接下來的第二章節我們會討論伺服器的 HTTP/RTSP 協定，內容包含伺服器與用戶端互動的流程。第三章為伺服器多工型態的探討，以及對 Iterative/Concurrent Server 的分別逐一介紹。第四章為重點所在，主要探討伺服器如何分配資源來傳送封包給多個用戶。第五章為結論。第六章為附錄，包含了 ffserver 與 live555 在 windows 下編譯，以及我們如何透過播放器的設定使得這兩套伺服器能正常的動作。

第二章　串流伺服器介紹

串流伺服器可以概分為循序式串流(Progressive streaming)以及即時式串流(Real time streaming) 傳輸，常見到的 HTTP Server 便屬於循序式傳輸，而 RTSP Server 便屬於即時式傳輸。兩者最大的差別是 RTSP Server 可播放已下載影音檔任意時間點的內容，而 HTTP Server 只能播放目前正下載的內容，以下將會更詳細介紹兩者的差異。

2.1　循序式串流伺服器 (HTTP Streaming Server)

採用 HTTP 協定最大的優勢為封包可以穿越防火牆的阻擋，並且一般的使用者可以直接使用如微軟的 Internet Explorer 瀏覽器來接收來自伺服器的影音封包，HTTP Methods 介紹如下表 2.1 所示：

HTTP Methods	Explain
OPTION	request for information about the communication options available on the request/response chain
GET	retrieve information
HEAD	retrieve information (test hypertext links for validity, accessibility, and recent modification)
POST	subordinate to a directory, newsgroup, database...
PUT	store entity
DELETE	delete entity
TRACE	see what is being received at the other end of the request chain

表 2. 1　HTTP　Methods

(註)上述的 method 在程式中可想成是一個函數的名稱，也就是在伺服器在解析出封包中所包含的信令資訊後，伺服器便會執行信令(text-based)所代表的函數內容，完成用戶端所要求的動作，而這些動作便是我們要去撰寫的函數程式內容。

2.2 即時式串流伺服器(RTSP Streaming Server)

即時式串流最大的優點是用戶可以快轉/迴轉去觀看已經下載完成的片段內容，RTSP Methods 介紹如下表 2.2 所示：

RTSP Methods	Explain
OPTION	Get available methods
Setup	Establish transport
Announce	Change description of media objects
Describe	Get (low-level) description of media objects
Play	Start to send packets to clients
Record	Start Recording
Redirect	Redirect client to new server
Pause	Halt delivery, but keep state and server resources allocated
Set_parameter	Device or encoding controls
Teardown	Removes state and free all resources

表 2. 2 RTSP Methods

2.3 循序/即時式伺服器比較

大部分的優缺點比較我們已於 1.2 章節(研究背景與動機)中討論過，在此僅做簡單的分析比較。

Server Type	Pros	Cons
HTTP Streaming	1. 封包不受防火牆的影響 2. 設計/操作簡單，使用者可直接以 IE 或 FireFox 瀏覽器連線觀賞之。 3. 適合文字/圖片的傳輸。	1. 不可即時快/迴轉觀賞。
RTSP Streaming	1. 可即時快/迴轉觀賞。 2. 適合聲音和影像的傳輸。	1. 網路擁塞時易影響傳輸品質。 2. 封包會受到防火牆的阻擋。 3. 設計/操作略微複雜，使用者通常需安裝特定的 RTSP串流播放器。

表 2.3　循序/即時式伺服器分析比較

2.4　串流系統架構

一個完整的串流系統架構一般由三個部分組成，其一是影音的壓縮編解碼；其二是封包的封裝演算法，也就是伺服器必須適當地將影音來源訊框(frame)分裝在每一個封包中，而RFC對於每一種影音分裝(packetization)也都有標準的建議，例如mpeg1便是參照RFC-2250的標準(註1)；其三是網路程式設計，此部分包含了封包的傳送、信令協定的溝通以及多工串流時間控制。

上述第三部分中的**多工串流時間控制**便是本論文主要探討的部分，該部分我們留至第四章再進行探討，接下來 2.5 章節我們便針對 ffserver 軟件做簡單的介紹，該軟件如我們在 2.1 章節提到的是一個循序式的串流伺服器 (HTTP Streaming Server)。

2.5 FFserver 串流伺服器的組成與多工的型態

在我們要提 FFserver 之前首先介紹一下 FFmpeg(註 2)，FFmpeg 是一多功能的編解碼(Transcoding)軟體，可以用於影像流與聲音流的分離、轉換、編碼以及解碼，主要由以下部份組成：

- ✧ FFmpeg；是個轉碼工具，也就是將某一個影音格式轉碼成另一個影音格式，例如 ffmpeg –i inputfile.mpg outputfile.mp4 就是將一個.mpg 檔轉換成.mp4 檔。
- ✧ FFserver；是個串流伺服器，藉由 HTTP 與 RTSP 的溝通協定進行即時串流。
- ✧ FFplay ；是一個用 FFmpeg 函式庫所開發出來的多媒體播放器。
- ✧ Libavocode 是 FFmpeg 所提供的函式庫，幾乎涵蓋了所有影音格式的編碼及解碼的功能，程式開發者只要將此函式庫引入專案中，即可讓使用其函式庫中所提供的編碼/解碼的功能。
- ✧ Libavformat 也是 FFmpeg 所提供的函式庫，利用這個函式庫可以將聲音流與影像流從檔案中抽取出來或是將壓縮過的聲音流與影像流結合成其它格

式的檔案。

2.5.1 FFserver串流伺服器的溝通協定

串流伺服器的組成如我們之前提到一樣有三大部分，第一部份的影音壓縮編解碼即是上述的FFmpeg；第二部分封包的封裝演算法即是FFserver；第三部分是網路程式設計，關於BSD Socket API (Socket、Bind、Listen、Accept、Connect、Read、Write、Close)部份請參考2.5.2章節。

下圖2.2所示為整個FFserver 所應用到的TCP Socket API（傳送RTSP命令）；圖2.3則為所應用到的UDP Socket API（傳送RTP封包），而串流伺服器一開始會建立一個TCP模式的RTSP Socket，等到用戶端Connect()連線成功後即進入RTSP信令的解析以及執行，等到收到用戶端發送RTSP PLAY信令後便以UDP模式建立一個RTP Socket來完成封包的傳送。

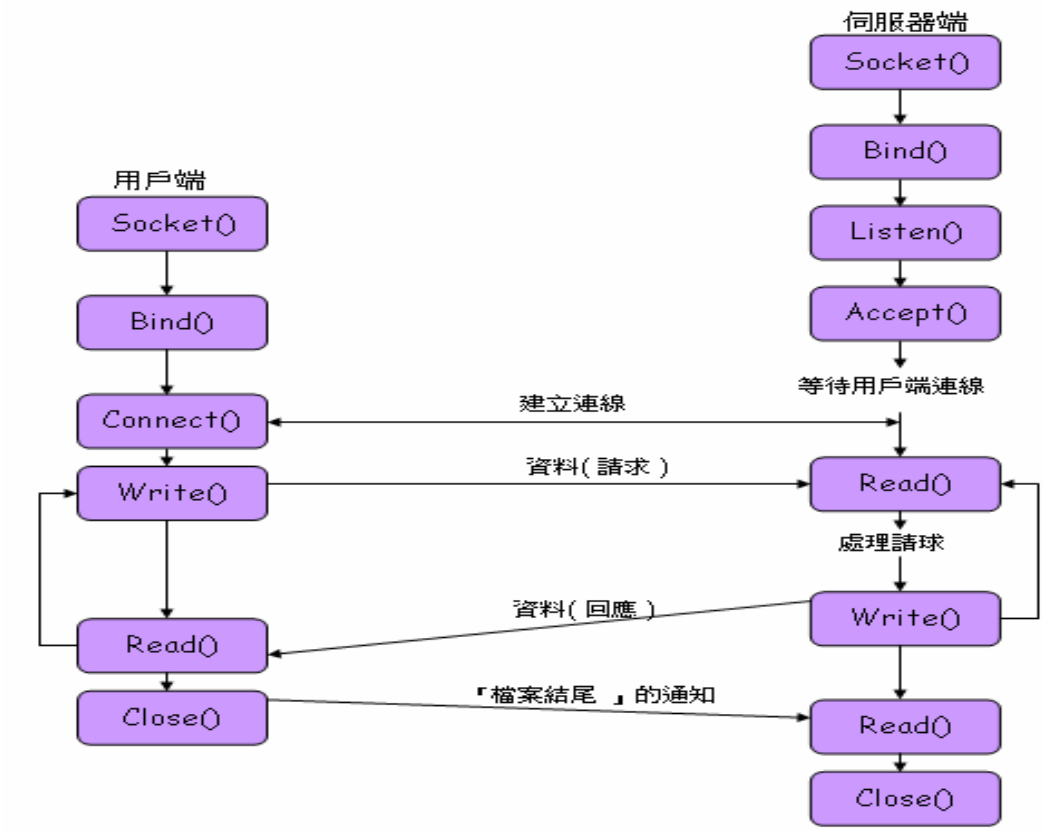


圖 2. 1 TCP Socket API flowchart

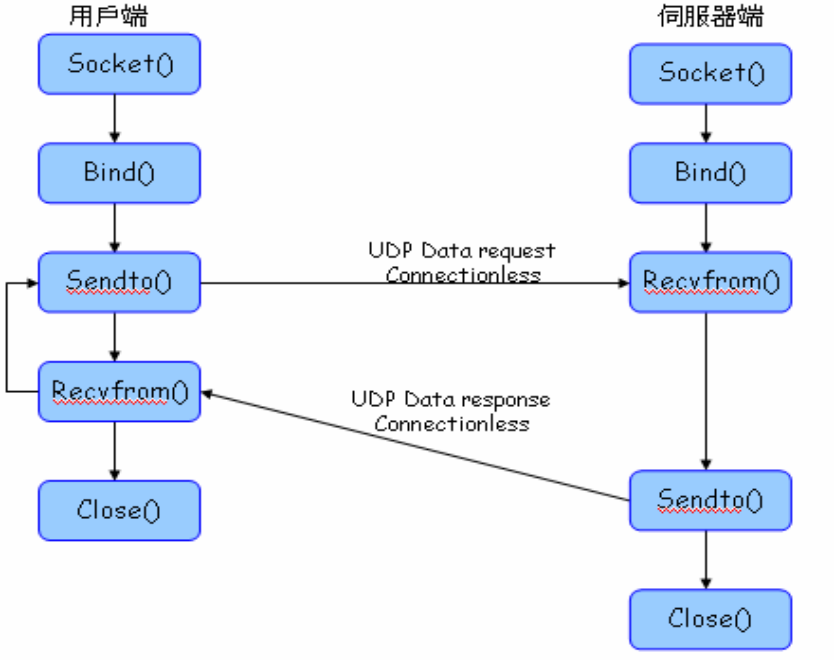


圖 2. 2 UDP Socket API flowchart

2.5.2 Socket 建立與連線的基本程序

如果兩個 processes 想要以 Client-Server 架構透過 socket 互通訊息，則此兩者必須依循各自的基本程序，才能成功建立連線與接收訊息。如下程序為一個 Server 要接受來自 Client 端的連線請求，以及順利地執行後續的接送資料之工作：

- ✧ 呼叫 Socket；建立一個 communication endpoint 。
- ✧ 呼叫 Bind；把 IP address 與 Port number 綁到 Socket 身上。
- ✧ 呼叫 Listen；命令 Socket 進入 Listening state，以監聽是否有 Client 提出 connection request。
- ✧ 當發現有 client 提出 connection request，呼叫 Accept 來接收連線要求，並建立一個新的，處於 connected state 的 Socket。
- ✧ 接下來進入傳送或是接收資料的 loop；透過 connected socket，呼叫 send/write，來傳送或是接收資料。
- ✧ 結束連線；呼叫 close 關閉 Listening socket 與 connected socket。

FFserver 在 socket 連線成功之後尚沒辦法將存在硬碟的資料傳出去，因為我們要傳

的檔案是存放在硬碟的某個地方，所以必須先把特定影音格式的資料解析(parse)出來進而傳到 Buffer 中，接下來在 Buffer 中針對 Parser 已經解析出來的 frame 做封包封裝的動作，同時將每一個 packet payload 加上 header，如此便是一個 RTP packet，最後便是呼叫 BSD Socket API 將 RTP packet 加上 UDP/TCP 以及 IP header，形成一個可在網路上傳送的結構化 IP packet。

以上便是串流伺服器的簡介，關於封包封裝傳送以及多工的部分我們在第四章中會有更詳細的解說。

第三章　伺服器多工型態

在第二章節中已介紹過串流伺服器的種類，接下來我們要探討的是在串流伺服器上所應用到的多工型態。**多工**，其觀念是一個串流伺服器可以**同時**服務多個用戶端的連線需求，並且**立即回傳**影音或文字封包。追根究底來說，伺服器在傳送給 A、B 用戶的封包間仍會有一段**時間間隔(Time Latency)**，但因時間間格太小，因此我們感覺起來就像伺服器會分身一樣，同時服務著 A、B 用戶。

而伺服器要達到多工較常見的組合型態有兩種，第一種是使用 **Fork+Select** 函數，第二種是使用 **Thread+Select** 函數，基本上以上兩種都是 **Multi-Process** ，也就是 Concurrent Server 的應用。另一種較不常見的是 **Single-Process** 的應用，這通常也就是指 Iterative Server，該應用除了使用 Select 函數以外還必須自行實作**時間控制機制**來分配伺服器對每一個用戶的服務時間。本章除了介紹上述三種多工型態外，也對 Iterative/Concurrent Server 的分別逐一介紹。

3.1 Iterative Server

Iterative Server 通常只會有一個行程(Single-Process)，因此該行程需同時擔任接待以及服務的工作，Live555 Streaming Server(以下簡稱 Live)的 RTSPServer 便是屬於此種串流架構。這種串流伺服器會由這個單一的行程來控制多個 sockets 資料的傳送與接收，如圖 3.1 左上方中最左邊的 socket(listening socket)負責監聽

是否有新的連結要求，並等待來自 client 端的 Connect()，Accept()之後便有獨立專屬的 connected socket 負責連線後續的工作。

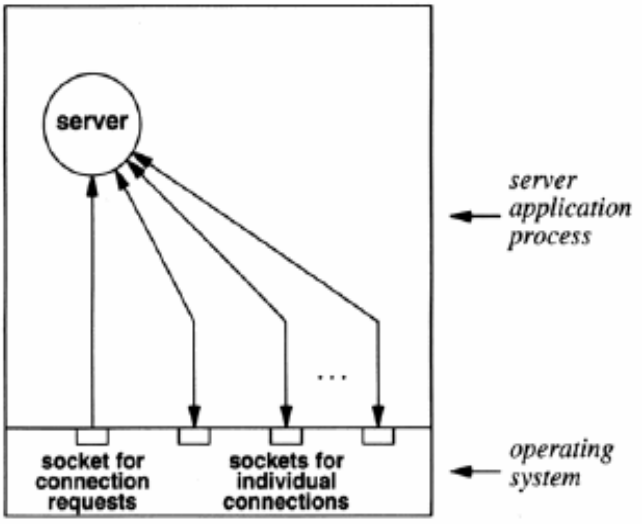


圖 3. 1　Iterative Servers

如同我們在以上所提到的，既然 Iterative Server 不能做到真正的多工，那為何基於 Iterative Server 架構的 Live Streaming Server 又可以做到多工呢？此部分的串流時間控制與時間分析在第四章中我們會有詳細的說明，此種多工架構的意義以及優缺點為何在第五章中會予以討論。而通常這種架構需要搭配 select()函數來使用，select()函數最大的好處是在沒有用戶的連線下，伺服器便不需要去 polling Socket I/O，浪費 CPU 資源。

3.1.1　Select 函數

Select 函數可以告知 kernel 它想監控的 socketfd，當 socketfd 所 index 的 socket 有事件發生時，kernel 會叫醒該行程來處理，省去平時沒有用戶連線時，CPU 去

polling Socket 的資源浪費。Select 函數可以監控的事件有三種，分別是『讀取』、『寫入』以及『例外』，函數敘述如下：

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

以下表格 3.1 為 Select 的參數說明：

maxfd1	爲fd_set中的可加入的socket數目上限值，該數目通常爲最大的描述子加1。
readset	爲socketfd 的集合，該集合專門負責監控socket是否已可以讀取。
writeset	爲socketfd 的集合，該集合專門負責監控socket是否已可以寫入。
exceptset	爲socketfd的集合，該集合專門負責監控socket是否有例外發生。
timeout	timeout會告訴Kernel要花多少時間來等待其中一個指定的描述子變成就緒。

表 3. 1 Select 函數裡的參數

[註：第二個至第四個參數，都是一個 data type 為 fd_set 的 pointer，這三個參數所指向的物件，其內容為所對應的 event 有所回應的 file descriptors。如果我們對其中一個的 event 沒有興趣或是不想讓這個 event 有所回應時，可將參數的值設定為 NULL 即可。]

以下表 3.2 為 FD_SET 巨集種類：

FD_SET	Describe
void FD_SET(int fd, fd_set *fdset);	用來將某個 file descriptor加入某個set中
void FD_CLR(int fd, fd_set *fdset);	用來檢測某個 file descriptor，從某個set中移除
void FD_ISSET(int fd, fd_set *fdset);	用來檢測某個 file descriptor，是否還存在於某個set中
void FD_ZERO(fd_set *fdset);	用來清除 set 的內容

表 3. 2　FD_SET Marco

3.1.2 Single Process Iterative Server

下圖 3.2 為 Single Process Iterative Server 的流程圖。在 **Iterative Server** 中，Listening socket 是專門負責監聽 client 的連線要求，而 Connected socket 則是已經建立好固定的連線，透過 Select 函數與巨集(詳細請參考 3.1.1 章節)，使得 **Iterative Server** 可以不斷地或是定期地去查詢是否有新的連線請求(新的用戶連線)，或是舊用戶的控制信令(RTSP 信令)。

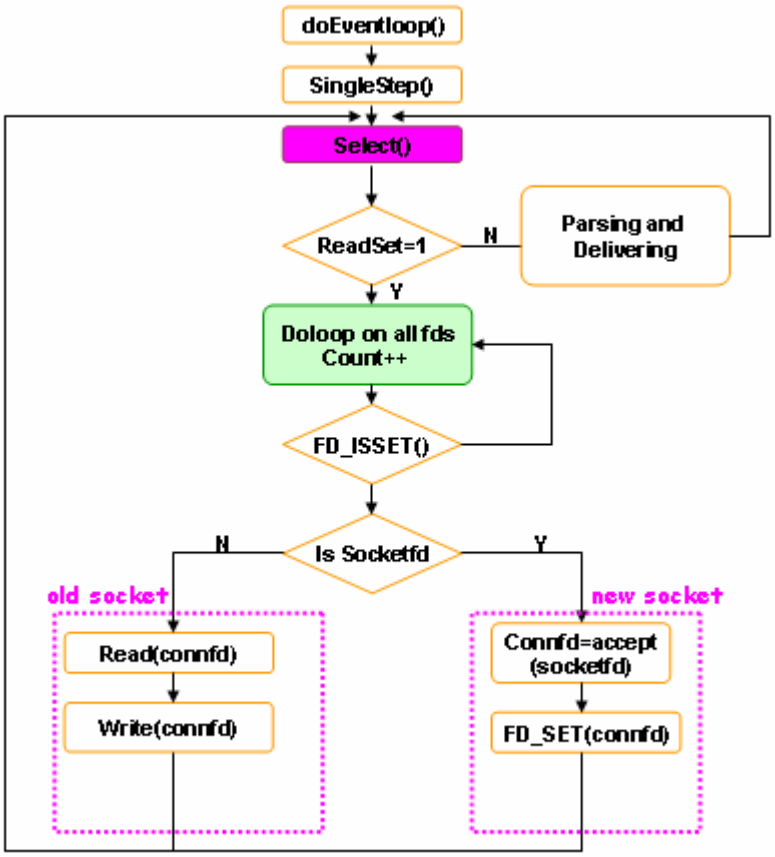


圖 3. 2　Single Process Iterative Server 流程圖

如圖 3.2 所示，Live 的 **Iterative Server** 原則上是一直在 DoEventLoop 的迴圈中執行，新用戶連線(new socket)會把建立好的 connected socket 加入 select 所監控的 set(集合)中；接下來當舊用戶的 connected socket 有 RTSP 的信令傳送時，kernel 變會喚醒 select，行程便繼續執行下去(old socket)。原則上就分為下列兩種的情況：

1. 尚未連線的 client 所提出的要求(如上圖 3. 3 new socket)。

2. 已連線的 client 所提出的服務請求(如上圖 3. 4 old socket)。

如果是尚未連線的要求，**Single Process Iterative Server** 會呼叫 Accept()來接受連線的要求外，還必須把 connected socket 加入欲監控的 set 中(圖 3.2 的 FD_Set(connfd))，使 connected socket 成為 select 的集合。而 **Iterative Server 會在 old socket 流程中**，藉由呼叫 BSD Socket API 的 Read/ReciveFrom、Write/SendTo 函數來進行與 client 間的封包收送的工作。

3.1.2.1 fd_set

在 Select 的引數中我們會看到 Set(fd_set)，實際上這個 set 是一個 bit array，其主要的功能是用來標示那些 sockets 是程式所想要去處理的對象，藉由不同的 set 來區分 event 的種類，並且根據 event 的種類去服務它的要求。從 Select 函數中的引數我們可以知道，它包含三個 set，用途參考下表 3.3 所示 ：

fd_set	Describe
read set	當single process server 對發生在某個socket身上的 new data event 有興趣，就將該socket加入read set 。
write set	當single process server 對發生在某個socket身上的 Buffer is available event 有興趣，就將該socket加入write set 。
exception set	當single process server 對發生在某個socket身上的 Error event 有興趣，就將該socket加入exception set 。

表 3. 3 根據 event 的種類區分 fd_set

3.1.2.2 Select 函數虛擬碼範例

fd_set fReadSet； //宣告 fReadSet 的 data type 為 fd_set

int max_fd； //用來記錄所有 sockets 當中，其值最大者

socket_fd = socket(…)； //Construct sockets

bind(socket_fd ,…)；

listen(socket_fd, …)；

FD_ZERO(&fReadSet)； //對 fReadSet 執行清除的工作

FD_SET(socket_fd , &fReadSet)；//監聽 client 連線請求的 socket 加入 fReadSet 中

/*目前程式中只有 listening state 的 socket，所以 max_fd 的值就是這個 listening socket 的值*/

max_fd = socket_fd；

while(1){

fd_set readset = fReadSet；//就是這個 listening socket 的值。

/*select 函數中所呼叫的是 read 的事件，故其它*writefds, *exceptfds 都為 NULL，而 tv_timeToDelay 預設值為一百萬秒，為 struct *timeval 型態所宣告變數的最大允許值。

故如果沒有新用戶連線的話，程式便會 block 在此點；倘若必須持續對舊用戶傳送封包的話，在圖 3.2 的 old socket 中，程式必須設定 tv_timeToDelay 為 0，如此程式的流程才得以繼續向下執行。

等到不需要對舊用戶傳送封包時，程式便恢復 tv_timeToDelay 為一百萬秒的設定*/

int selectResult = select(max_fd + 1,&readfds , NULL , NULL , &tv_timeToDelay)；

/* FD_ISSET 由 fd 0 開始去 scan 一直到 max_fd 為止，目的是先確認 listening socket 是否因為有新用戶連線要建立 connected fd；否則便由之前的 connected fd 逐一檢視，找出欲進行 RTSP 信令或是 RTP 封包傳送的用戶，其流程請參考圖 **3. 5***/

```
//Below pseudo codes are under while-loop
for(int i=0；i < max_fd；i ++)
{
    if(FD_ISSET(i , &readSet)= = 1)
    {
        if (i= = socketfd) // new connection
        {
            Connfd = accept(socketfd …)；
            FD_SET(connfd , &fReadSet)  ；
            if(connfd > max_fd)
                max_fd = connfd  ；
        }
        else
        {
            //TODO：RTSP negotiation and creates UDP socket for RTP
            (……. ……. ……. …….)
         }
            //TODO  ：send RTP packet to the exactly fd
            (……. ……. ……. ……. …….)
    }//end if FD_ISSET
} //end loop on all fds
```

以下為一個 Client 連接後，進入 Select 函數所產生的程序示意圖：

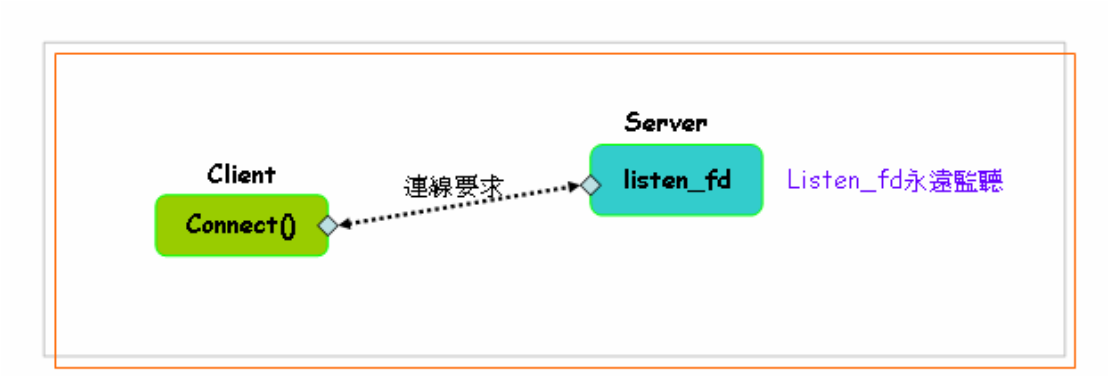


圖 3. 6　Step1 : Select() in Client - Server

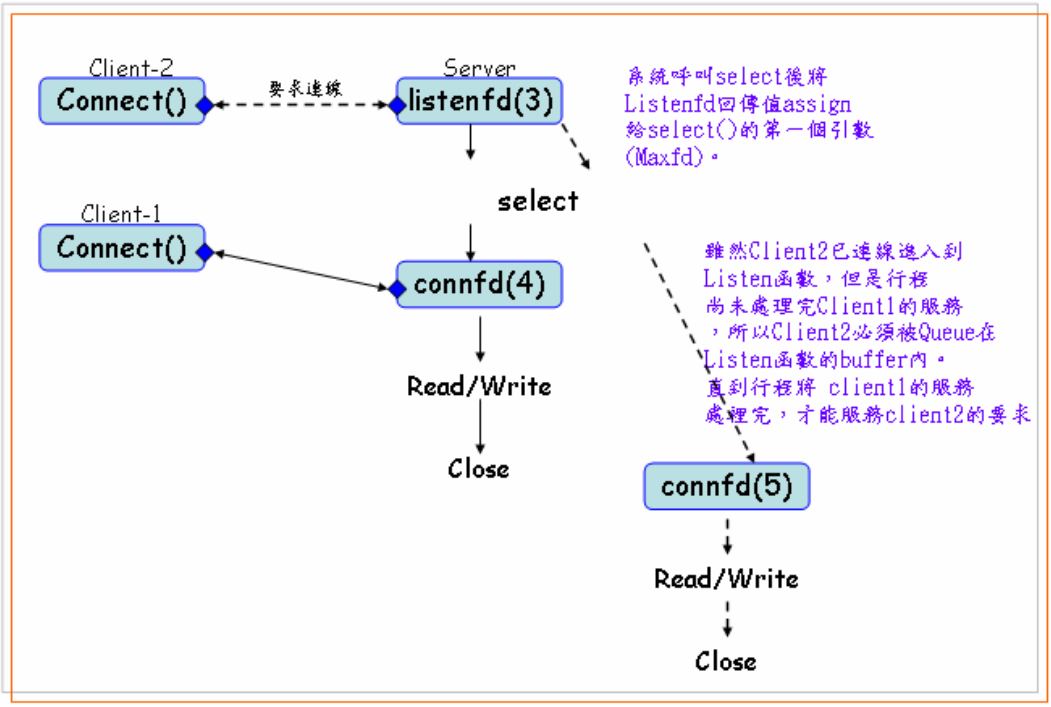


圖 3. 7　Step2 ：Select() in Client-Server

3.2 Concurrent Server

Concurrent Server 的意思是主行程(master Process)只負責接待但不負責服務,服務的工作是交由子行程(slave Process)來負責;在設計一個 Concurrent Server 時最簡單常見到的是透過 fork 這個 system call，由 Parent Process 複製出一個 Child Process，由 Child Process 來完成後續的服務工作，包含了 RTSP 信令的溝通以及 RTP 封包的傳遞；而 Parent Process 只負責監聽與接受連線請求，示意圖如下 3.1 所示。

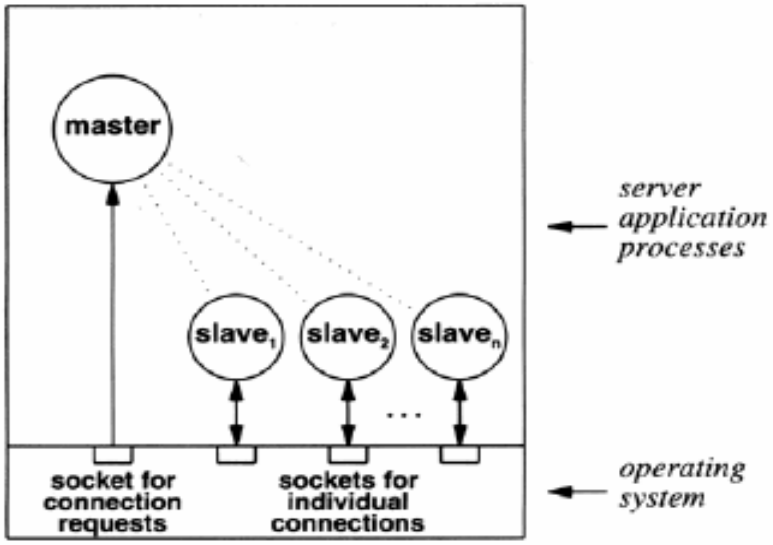


圖 3. 8　Concurrent Server 示意圖

當我們利用 fork()在建立 Concurrent Server 時，由於一個 Parent Process 複製出一個 Child Process 之後，必須關閉 Parent Process 的 connected socket (connfd)，因為它只負責監聽與接受連接的請求，因此這個 connected socket(connfd)對它而言是沒有用處的；反之分叉出來的 Child Process 則必須關掉 listening socket，理由如上段所述。

3.2.1 Fork 函數

fork()通常是配合 Blocking socket 來使用，fork()會在程式中產生一個新的子行程，並複製主行程的資料與堆疊空間，繼承主行程的 UID、GID、環境變數、已開啟的檔案代碼、工作目錄和資源限制等。如果 fork 成功則在主行程會傳回新建立的子行程代碼(PID) ； 若在子行程成功會傳回 0，若失敗則傳回-1。

3.2.1.1 fork 函數虛擬程式範例

```
int pid ；
socket_fd = socket(…) ；//construct socket
bind(socket_fd ,…) ；
listen_fd = listen(socket_fd, …)；
while(1){
    connfd = accept(socket_fd,…)；
    //Parent Process
    if (pid = fork() > 0 ) // pid > 0 為 Parent Process
    {
        close( connfd ,…)；
        continue；//if fork is ok , still in the while loop
     }
    if(pid = 0 ) // pid = 0 為 Child Process
    {
        close(listen_fd , …)；
        //TODO：RTSP negotiation and send RTP packet
        ……. ……. ……. ……. ……. …….
```

```
        }
}// end while-loop
```

下圖 3.6～3.8 為一個 Client 連接到 Concurrent Server(Parent Process)後，進入 fork()函數所產生 Child Process 的示意圖。

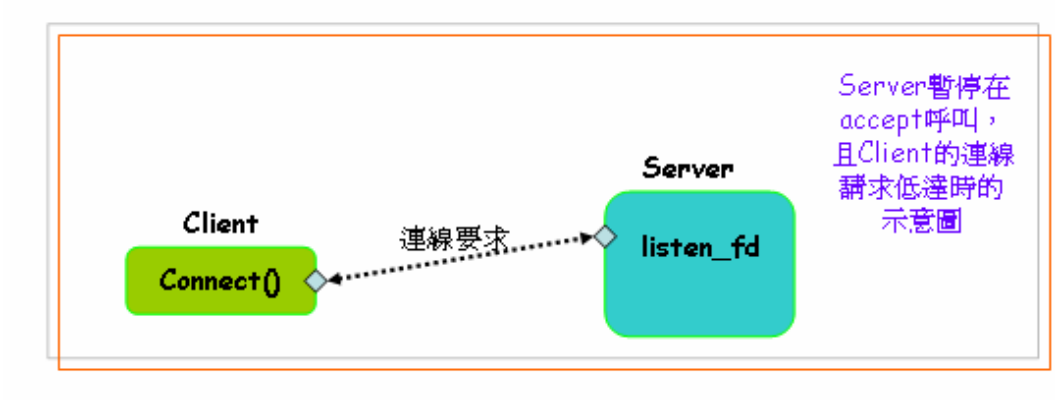


圖 3. 9　Step1：Fork in Client-Server

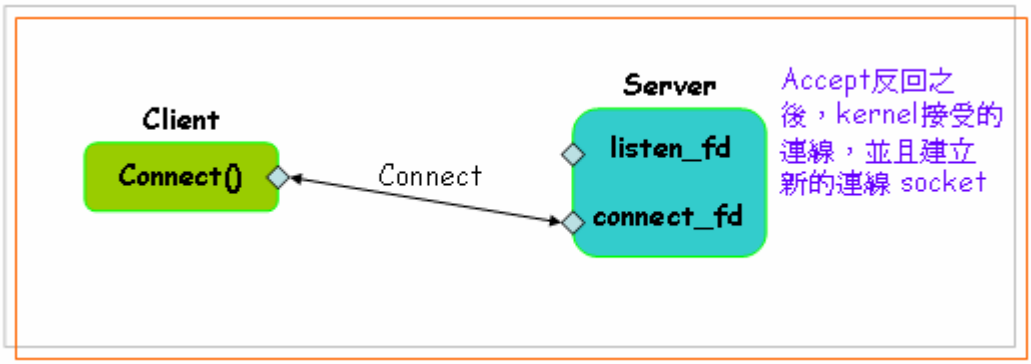


圖 3. 10　Step2：　Fork in Client-Server

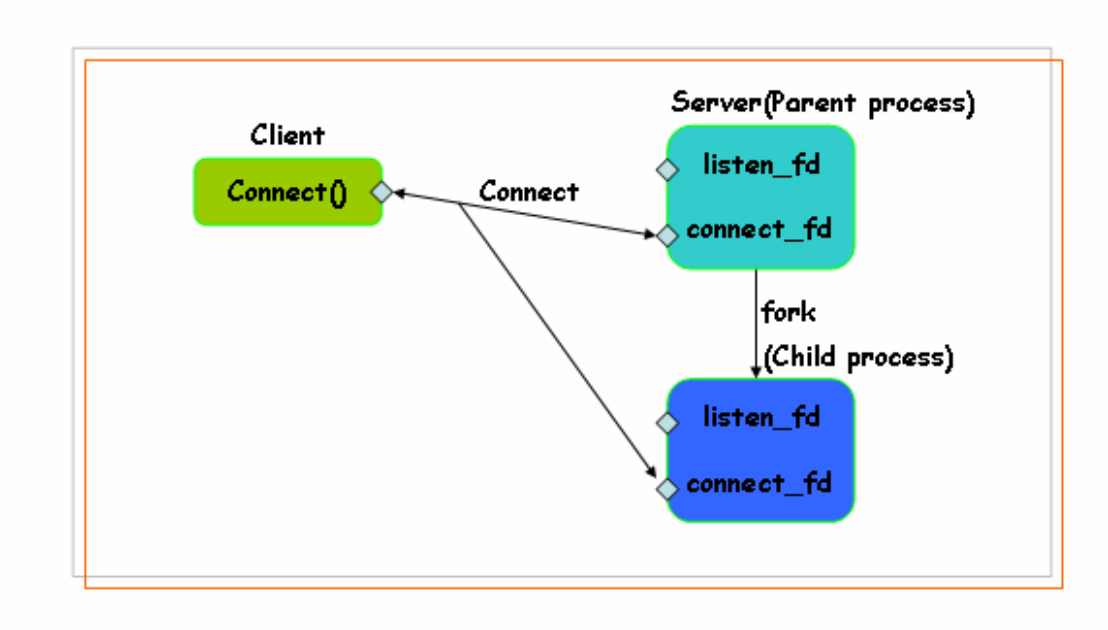


圖 3. 11　Step3： Fork in Client-Server

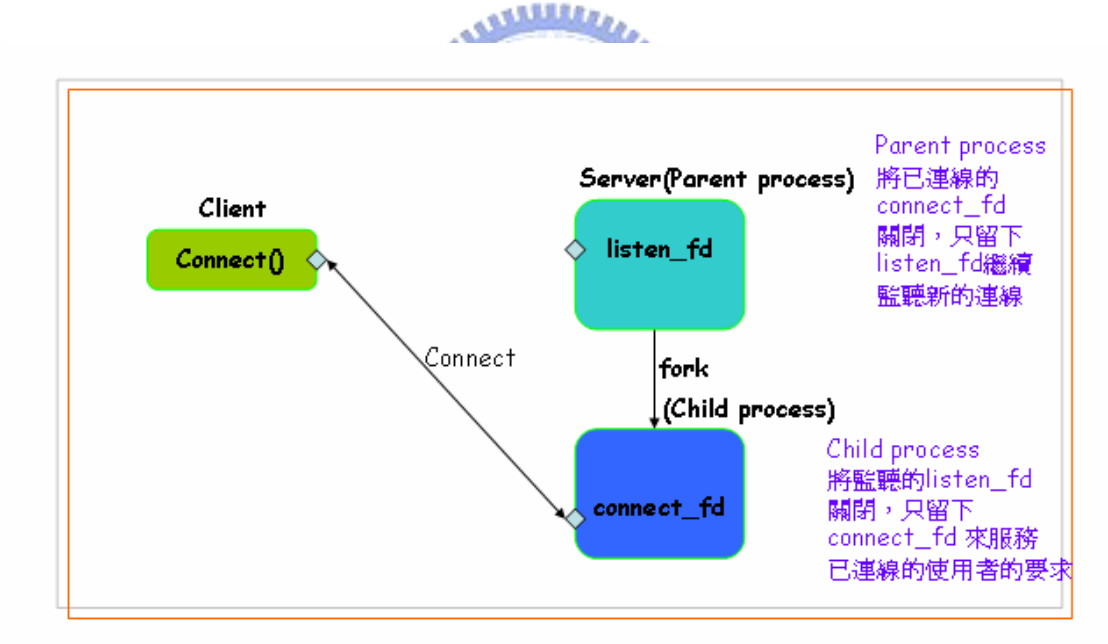


圖 3. 12　Step4： Fork in Client-Server

Fork 在使用上可說是非常方便，但在使用上需特別小心 Zombie Process 以及 Memory Leakage 的回收處理。

3.2.2 Thread

Thread 以及 Fork 基本上都是用來達成 Multi-Process 的函數，但是 fork 在使用上卻有兩個較嚴重的問題：

1. fork 時要將 Parent Process 的記憶體內容 copy 給 Child Process，包含所有的 socket descriptors 都要給 Child Process，相當耗費記憶體空間。
2. fork 之後，如果 Parent Process 與 Child Process 之間要傳遞資訊，必須透過 IPC 來達成，所以資料在行程間的傳遞亦會花較多的時間。

如果我們使用 thread 就可以改善這兩個問題，因為一個 Process 內的所有 thread 都會享有一個相同的全域記憶體空間，故可以節省記憶體的使用，接下來便說明 Concurrent Server 在使用 thread 上的細節。

3.2.2.1 thread 函數的用途

thread 我們稱為輕量級的行程『lightweight Process』，而每個 thread 有它各自的 thread ID、暫存器、程式計數器(program counter) 、堆疊指標(stack pointer)以及優先權。而在一個行程內的 thread 之間可以享有全域變數外，還可共享以下幾點：

1. 行程的指令
2. 開啟的檔案，如描述子(descriptors)
3. 信號處理的函式和信號的處理，如 SIGNAL
4. 目前工作的目錄
5. 使用者的 ID 與 群組的 ID

thread 函數敘述如下：

int pthread_create (pthread_t *tid , const pthread_attr_t *attr ,

void * (*func) (void *), void *arg) ；

以下表格 3.4 為 thread 的參數說明：

pthread_t *tid,	函式成功地建立新的 thread 時，新 thread 的 ID 透過 tid 指標傳回來
pthread_t attr_t *attr,	用來設定 thread 的優先權，初始的堆疊大小
void * (*func) (void *),	建立新的 thread 時，指定函數讓它執行
void * arg ,	以指標的參數arg當作函式(func)呼叫時的唯一參數

表 3. 4 Thread 參數說明

3.2.2.2 thread 函數虛擬程式範例

static void * doit(void *arg)； //宣告 thread function phototype

socket_fd = socket(…)；//Construct socket

bind(socket_fd ,…)；

listen_fd = listen(socket_fd, …)；

while(1) {

fd_set readset = fReadSet ；

int selectResult = select(max_fd + 1,&readfds , NULL , NULL ,NULL)；

for(int i=0；i < max_fd；i ++)

{

if(FD_ISSET(i , &readSet)= = 1)

{

```
            if (i= = socketfd) // new connection
            {
            Connfd = accept(socketfd …)；
            FD_SET(connfd , &fReadSet)；
            if(connfd > max_fd)
            max_fd = connfd  ；
            }
            else
            {
                  pthread_t tid；
                  pthread_create(&tid, NULL, doit, (void* )iptr)；  //建立新的 thread
            }
      }//end FD_ISSET
} //end for loop
```

```
//doit 函數負責每個 connected fd 的封包傳送的工作
static void * doit(void *arg)
{
      connfd = *((int *) arg)；
      //TODO：RTSP negotiation and send RTP packet
      …….. …….. …….. …….. …….. ……..
      …….. …….. …….. …….. …….. ……..
      pthread_exit(NULL)；//當封包傳送完畢之後，呼叫此函數結束執行緒的執行
}//end while_loop
```

由上面的虛擬程式碼我們可以看到如何在 Concurrent Server 中使用 **pthread** 函數來建置執行緒，thread 在串流伺服器的應用層面很廣，也就是我們可以把特定的工作安排給特定的執行緒來執行;例如我們也可以將 Listening socket 的工作安排給 thread-A 來執行，Connected socket 後的工作安排給其他 thread 來執行，例如 RTSP Parser 的工作安排給 thread-B，封包的切割封裝與傳送交由 thread-C 來執行。

Thread 的原則是透過**適當的任務分配**來增加程式**執行的效率**(減少執行花費的總時間)，因此 Thread 之間的 IPC 問題便是設計多執行緒(Multi-thread)行程的關鍵所在，因為要 Thread『分工』很簡單，但要它們『合作』便很難；同時也因為 **shared memory** 的關係，因此每一個**串流引數的使用**以及 **Thread 的執行順序**都要特別注意，避免設計出一個不穩定的串流系統，增加日後 Maintenance 以及 Debugging 的負擔。

第四章　多工串流時間控制與數據分析

本章論述的重點在於伺服器如何同時服務多個用戶，也就是去研究 Iterative Server 如何做到『**多工**』，探討 Iterative Server 在 Single process 中的**多工時間控制機制**。

4.1　多工的串流時間控制

在發展串流多工系統的時候，時間控制是一項很關鍵的議題。隨著用戶連線數目的增加，伺服器仍必須提供穩定流暢的封包來源，因此對每一個用戶的時間控制便需能拿捏的恰到好處，否則可能出現 A 用戶的串流品質很好，但 B 用戶的品質卻是斷斷續續的現象。

也因為去安排每個串流行程的時間控制是一項很複雜的工作，因此如果以**設計方便**的層面來講，我們可以直接呼叫 fork / thread 函數，將每個行程間的控制時間交給 OS 來負責管理，減少 programming 的負擔。另一種 scheduling algorithm 則是基於 Single-process 之上，帶來的好處主要是 **Debugging** 的方便。因為基本上程式是依循著**單步執行**的原則，因此在程式的追蹤除錯上基本上都能夠發現問題所在。

反過來說，這也是 fork / thread 多行程(多緒程式)的問題所在。因為這些行

程都是由 OS 依當時 CPU loading 來分配系統資源給行程使用，因此每次引起伺服器執行出錯的地方都會有所不同，通常這些都是 IPC(Inter Process Communication)的問題，例如 shared memory、semaphore 以及 mutex 等等。而隨著用戶連線數目的增加，IPC 將會變成影響多工系統穩定性的 critical section。

本章接下來將分析 Live555 這套軟體的時間控制機制，瞭解其演算法的**依據為何**是我們的核心所在，同時因為時間控制演算法會牽涉到一些 Codec 的觀念，故相關的議題如 MPEG 我們也會略做簡介。

4.2 MPEG Introduction

MPEG-1 初制定時，其解像度只有 350＊240＊30，以此解像程度每當用於電腦螢幕上播放時效果不佳，故其並不適合做視訊傳播的應用，因此 ISO/IEC 在 1990 年又開始制定 MPEG-2 的視訊壓縮標準，增加了視訊應用範圍的彈性，其中包含了支援不同的影像的解析度、增強畫面比率化的計算、加強時間和空間的資料壓縮性能，最大的改善在於提高數位率和使用調整位元率(VBR) 的技術，主要的應用包含了數位電視視訊的傳播、高畫質數位電視(HDTV)、及數位儲存媒體的應用，例如 DVD(Digital Versatile Disk)。而 MPEG-2 壓縮標準最後於 1995 年完成。

4.2.1 Group of Pictures(GOP)

一個 GOP 是由許多的畫面組而成，如下圖 4.1 所示，以下對 I/P/B-picture 做簡要的說明。

I 畫面(Intra - Pictures)：即是一個 GOP 中最重要的畫面，因為其後的 P 畫面和 B 畫面都會參考它的資料來編解碼，因此在串流傳輸 I 畫面時需特別小心。例如我們可以在傳送 I 畫面時，改用 **TCP socket** 來傳送整個 I 畫面的數個封包，確保 I 畫面能正確的傳送到用戶端。

P 畫面(Predictive Coded Pictures)：P 畫面的編解碼主要參考為前一個 I 畫面或 P 畫面，而參考的位置以移動**預估所產生的移動向量來表示。**

B 畫面(Bidirectionally Predicted Pictures)：B 畫面在編解碼時，會使用到前面 I/P 畫面的資料。在此我們再次強調其顯示的順序與編碼的順序是不同的，因為後面我們會發現 Live555 的時間控制演算法的依據也是起源於此，示意圖如下 4.1 所示。

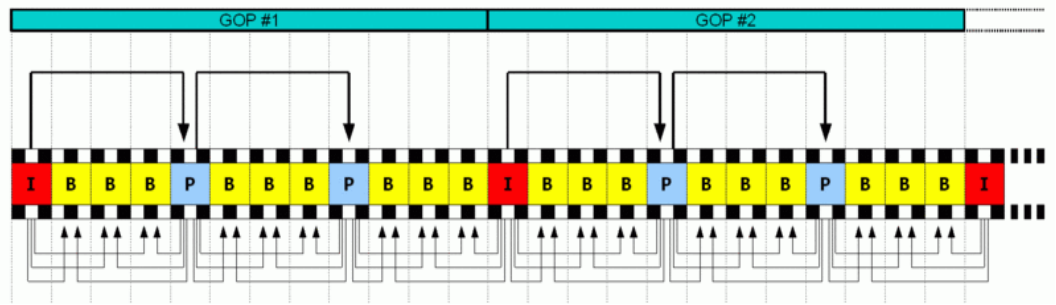


圖 4. 1　Group of picture

4.2.2　PTS　and　DTS

PTS（Presentation Time Stamp）是記錄播放的時問標記，也就是每個 frame 預計播放的時間點。如下圖 4.2 所示為 PTS 的示意圖；而 DTS(Decoding Time Stamp)則為記錄解碼的時間標記。如下圖 4.3 所示。

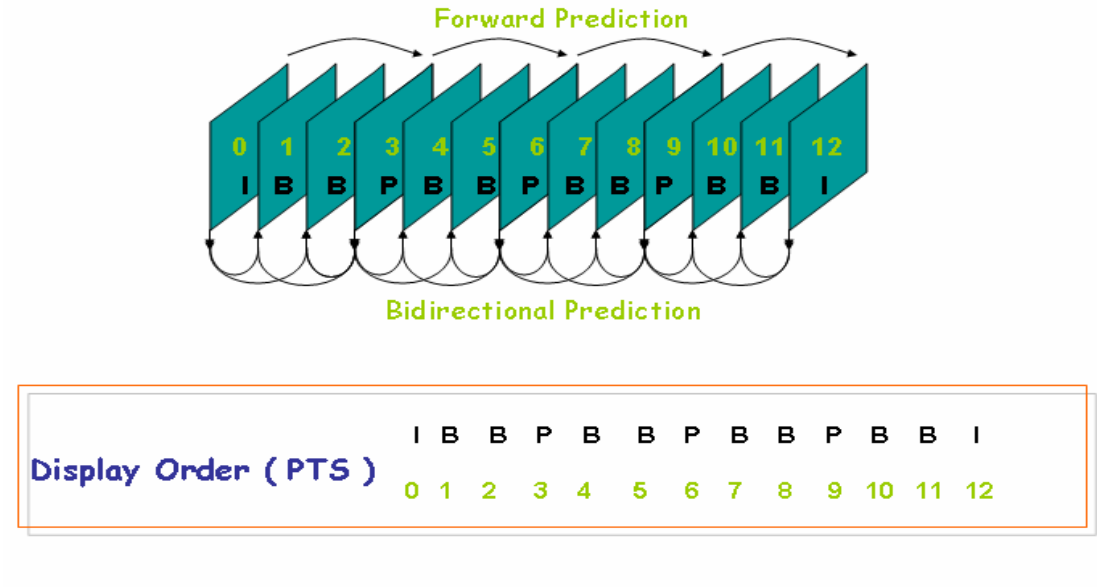


圖 4. 2　PTS　Display　Order

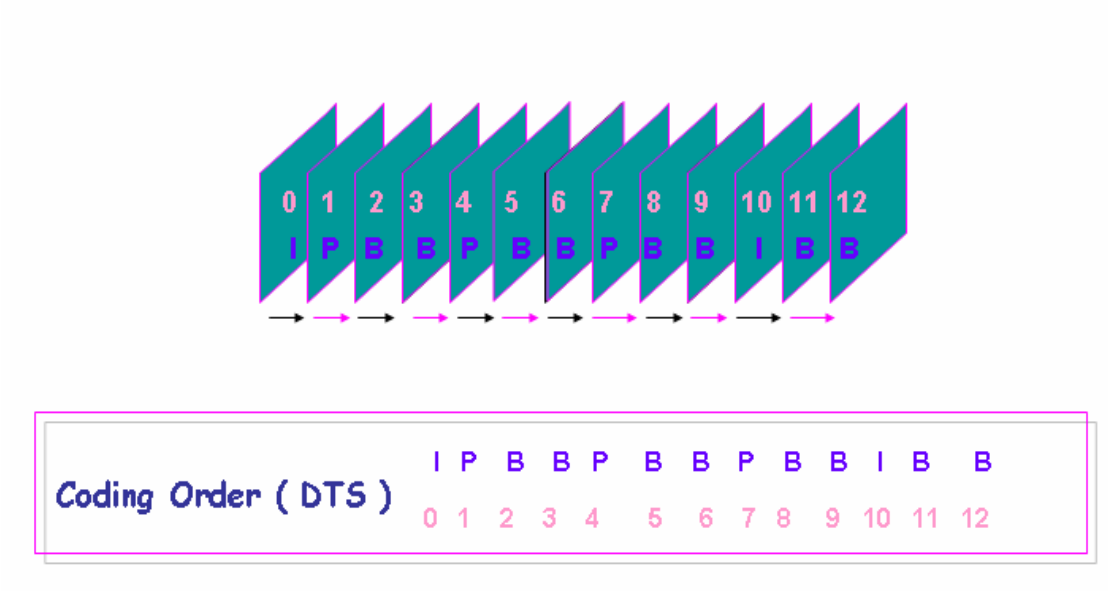


圖 4. 3　DTS　Coding　Order

如我們 4.1 章節所提到的，單一行程要能達到多工，我們就必須自行設計行程的時間控制演算法，我們知道 30 FPS(frame per second)代表每秒傳送 30 張 frame，平均起來一個 frame 的播放時間便是 1/30 秒。也就是說，假設第一個 frame 的播放時間的是 1/30 秒，第二個 frame 的播放時間便是 2/30 秒，……以此類推。

如果一個 frame 的大小超過 1500 Bytes(MTU：Maximum Transfer Unit)的話，意即該 frame 必須被切割成數個封包再傳出去。而由於 RTP 表頭中的 timestamp 便是記錄 frame 的播放時間，因此如果是同一個 frame 所切割出來的封包，它們的 timestamp 均會相同，而用戶端的播放器便是按照 RTP header 中的 timestamp 以及 sequence number 將封包排序後再予以播放。而 timestamp 便是由 PTS 所得

到的值(詳細說明請參考 4.3.1 章節)再乘上 90000(ex:MPEG2)即可得到。了解 PTS/DTS 的意義及用途後，我們接下來便可以進行時間控制演算法的分析。

4.3 單行程串流伺服器整個系統流程分析

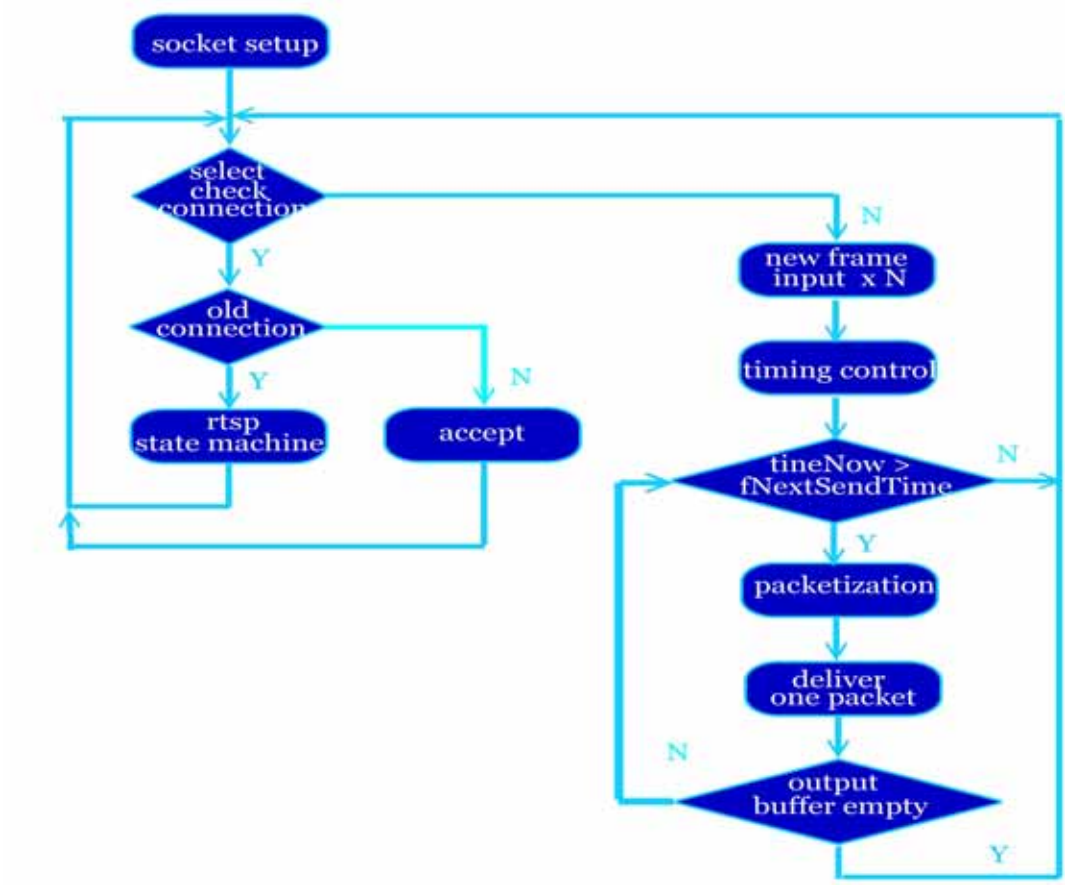


圖 4.4 單行程串流伺服器整個系統流程圖

單行程串流伺服器的流程圖如上圖 4.4 所示。一開始便是建立 socket，接著會使用 select 函式來監看是否有連線的要求，當有連線要求的時候，伺服器還要

去檢查這連線要求是舊連線或是新的連線，舊連線代表這個連線已經被伺服器接受(accept)過的連線，而新連線代表此連線是第一次連到伺服器，即尚未被伺服器接受過(accept)的連線。

如果是新連線的話，伺服器便會去接受(accept)此連線，並給這個新連線一個新的 socket descriptor；如果是舊連線的話，伺服器便會去解析(parse)此連線的 RTSP 信令，也就是伺服器會進入 RTSP state machine(狀態機)之中，例如在收到 RTSP『PLAY』的信令之後，伺服器便會進入『PLAY state』中，也就是開始去進行封包的發送。

當伺服器接受(accept)了一個新的連線或走完 RTSP state machine 後，伺服器便會 block 在 select 函式的所在並且等待新的連線要求。此外，select 函式並不是只有在連線要求的時候才會回傳(return)，當時間等到 timeout(select 函式第 5 個參數所標示的數值)select 函式亦會回傳。伺服器接下來該做的工作便是去做 parse 的動作。

伺服器便會對每一張 frame 計算其 PTS 和 fNextSendTime，並且依照 fNextSendTime 來傳送該 frame。當某個使用者的 frame 所切割出 payload 的 fNextSendTime 小於系統時間(Wall clock)時，意即該 payload 可以開始經過封裝

傳送的處理了，而在此需特別強調的是同一個 frame 所切割出來的 payload 都會具有同樣的 fNextSendTime。而本論文重點便是著墨在時間控制機制的部份。

但是 frame 在傳送之前必需經過切割封裝的處理，也就是基於 MTU(Max Transfer Unit)把 frame 切割成數個小 payload，並且把這些 payload 放在系統的 output buffer(輸出封包緩衝區)中去待命傳送。同時伺服器會去計算 timestamp，並且把完成的 RTP header 配置在 payload 的前端，如此便完成了一個 RTP packet 的產出。接下來藉由呼叫 BSD socket API 來替 RTP packet 加上 UDP 以及 IP header(由 OS kernel 負責)，完成整個封裝的動作。此時，一個可在網路上傳送的標準結構化封包(IP packet)便產生了。

當只有一個使用者時，伺服器在傳送完一個 IP 封包後，伺服器便會去檢查 output buffer 是否還有未處理的 payload 要出去，如果尚有 payload，伺服器便會將剩下來的 payload 傳送出去。

當有兩個使用者以上時，伺服器在傳送完一個 IP 封包後，伺服器便會去檢查 output buffer 是否還有其他使用者未處理的 payload 要傳送出去，如果尚有 payload，伺服器便去檢查該 payload 的 fNextSendTime。

當 fNextSendTime 小於系統時間(Wall clock)，伺服器便進行 payload 的封裝程序，同樣在封裝完成後便把該 IP 封包經由網路傳送至使用者端。但是 payload 的 fNextSendTime 未小於系統時間(Wall clock)，則伺服器便回到 select 函式；如果伺服器送完某個封包後，發現 output buffer 內已經沒有其它的封包存在了，伺服器亦會回到 select 函式，重新開始一個新的流程(迴圈)，這便是整個單行程串流伺服器的處理程序。

4.4．單行程多使用者的時間控制流程

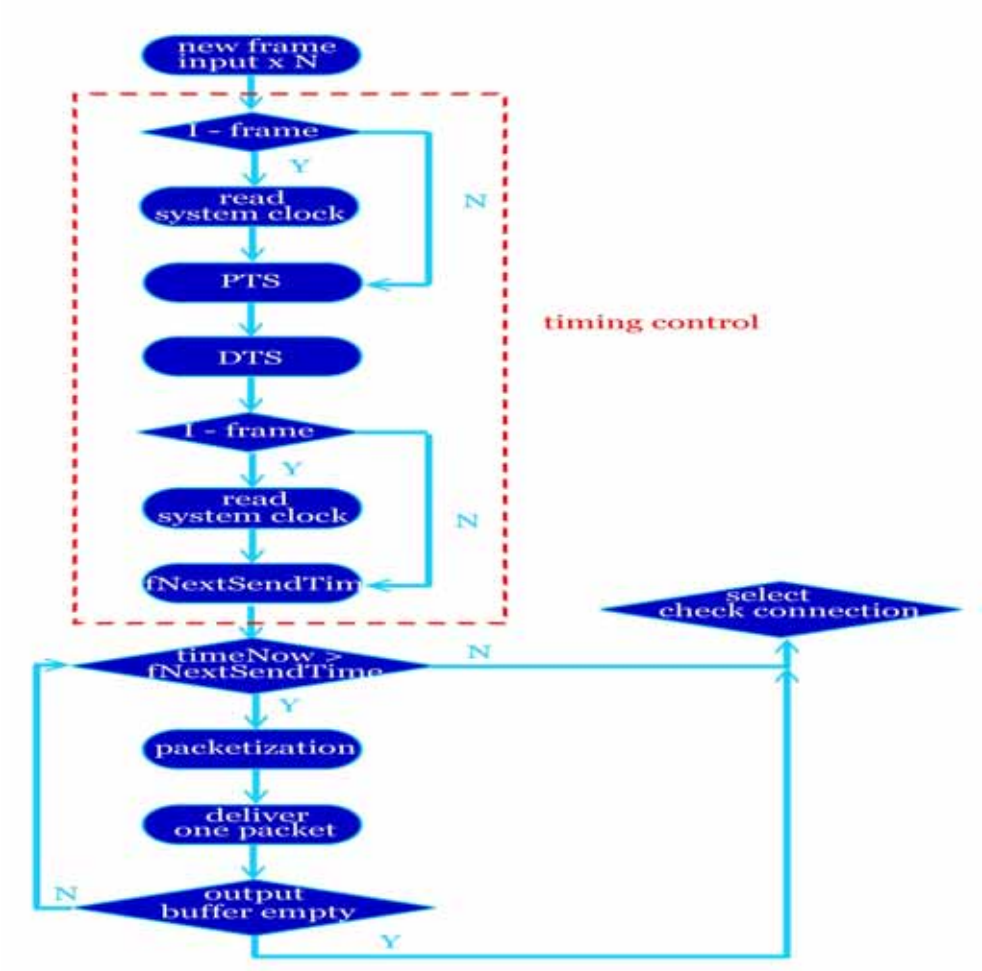


圖 4.5 單行程多使用者的時間控制流程圖

在單行程串流伺服器中，最重要的部分為傳送封包時的時間控制。封包的時間控制包含了許多封包時間的轉換機制， 如 parsing order 轉成 display order。

如圖 4.5 紅框內所示，而時間控制機制開始於伺服器對每一個使用者解析(parse)完一張 frame 之後，當伺服器進入時間控制的區塊(紅框 timing control 部份)時，伺服器會先計算各張 frame 的 PTS，在計算 PTS 時就需要一個時間轉換機制，而時間轉換的機制會在後面的 4.5.1 章節中說明。

計算完各張 frame 的 PTS 之後，伺服器又必須把 frame 的 PTS 轉成 DTS，這裡需要另一個時間轉換機制，同樣請參考 4.5.1 章節。接下來，伺服器便準備要把 payload 送去做 RTP 的包裝，但是 payload 何時該去做包裝，就必須要靠另一個時間參數 (fNextSendTime)來做判定。至於 fNextSendTime 是如何得到的，以及如何用此時間參數來掌握封包傳送的時間，我們會在 4.6.1 章節中說明。

4.5 時間控制演算法的分析

TimeStamp 欄位包含在 RTP 表頭檔中，且 TimeStamp 是由每一個 VOP 的 PTS 經過計算而得到的，然而在伺服器端檔案的 parsing 是以 DTS 的順序排列，為了計算出每一個封包的 timestamp 值，我們必須透過某些計算方式把 parsing order 轉成(display order)PTS，而這個計算方式由圖 4.7 t(i)函數運算而得到的。

4.5.1 單一用戶的訊框處理

下圖 4.6 是 parsing /display/sending order 的示意圖，經過壓縮後的影音多媒體便稱為 Elementary Stream，串流工作的開始首先便是去 parsing Elementary Stream，也就是把 Elementary Stream 中的 frame(如 I/P/B frames)給解析出來，其後便得到圖 4.6 中的 parsing order，由於 elementary stream 是以 DTS 的順序存放在硬碟裡，所以 parsing order 也是以 DTS 的順序排列。

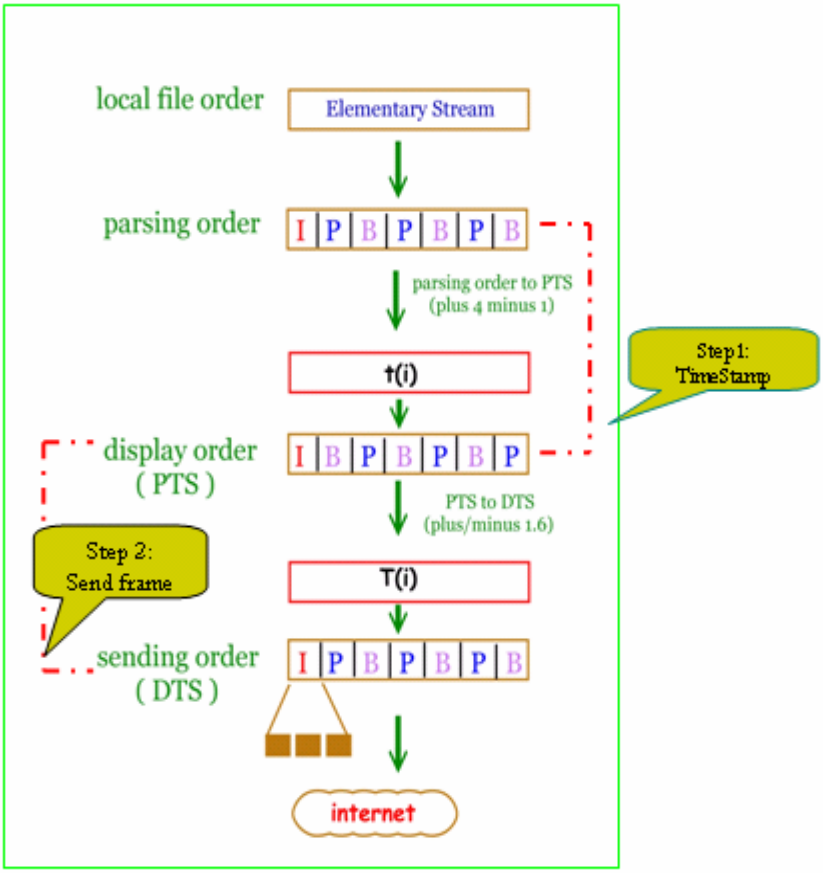


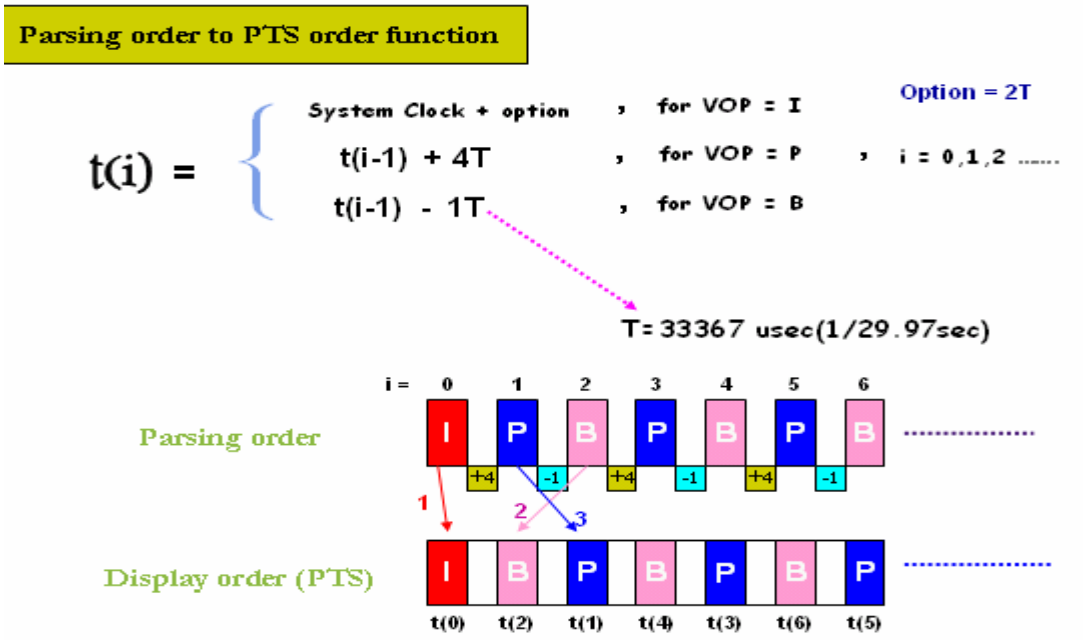
圖 4.6 parsing / display/sending order diagram

得到 parsing order 後，雖然我們可以直接對 frame 進行切割、封裝以及傳送的動作，但因為我們需要把多媒體影音檔包裝成 RTP 封包，所以必須填上 RTP

的 header，而其中的 Timestamp 欄位需要用到每一個 frame display order (PTS)的值，因此我們必須將 parsing order 轉成 display order。而這個轉換機制便是圖 4.6 中的 t(i)，運算式如圖 4.7 中所示(詳細請參考 **4.5.1.1 t(i) 函式計算**)。

接下來我們又要再次將 display order 轉成 sending order(DTS)，主要原因為 frame 必須要以 sending order(DTS)的排列順序來傳送，如此用戶端的播放器接收到 frame 時才能依序解出 P 與 B frame(請參考 4.2.2 PTS/DTS)。而 display order(PTS) 轉成 sending order (DTS)的機制是由上圖 4.8 中的 T(i) 函式所算出來的(4.5.1.2 T(i) **函式計算**)。

4.5.1.1 t(i) 函式計算



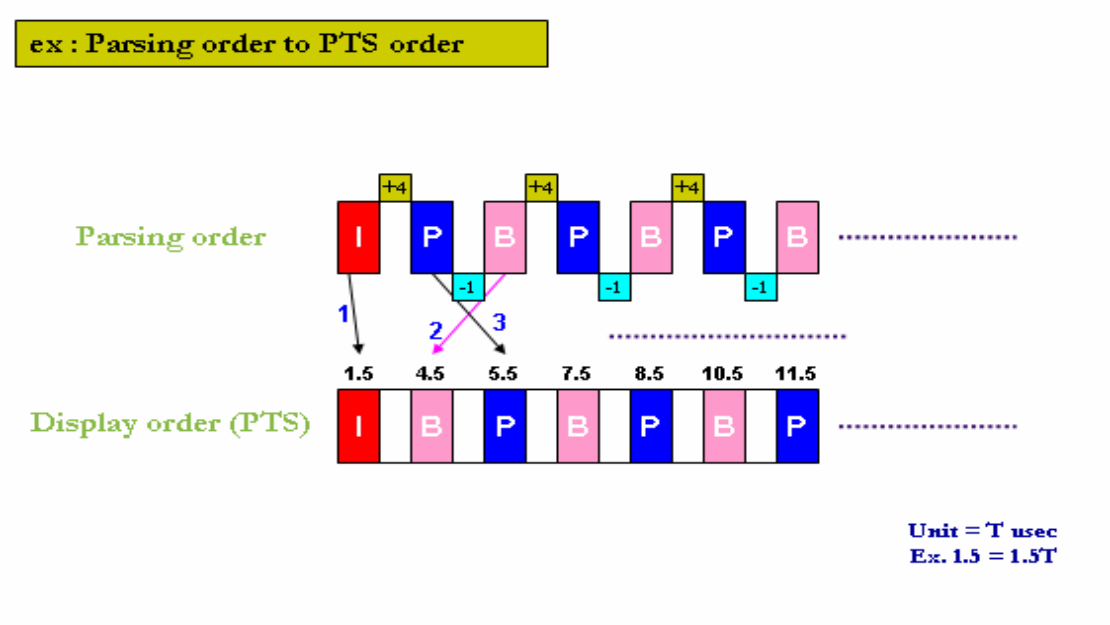
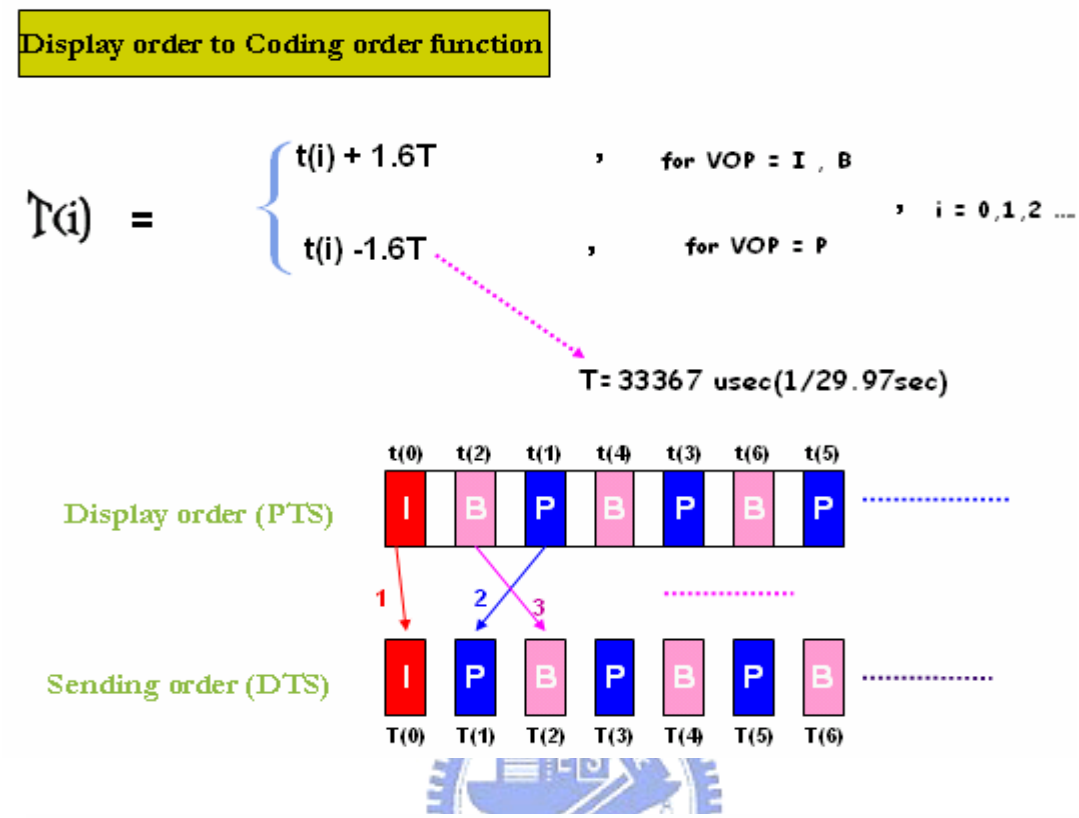


圖 4.7　t(i)函式運算

圖 4.7 中，由於在計算時間時需要以一個初始值做為基準來進行 frame 順序的排列，所以我們選取系統上的時間(System Clock，亦稱為 Wall clock)來當作 frame 順序比對及排列的時間基準點(Time Base)。

系統時間經過 t(i)函式的運算後便可得到各個 VOP 的 PTS，其中每一個 I-VOP　PTS 的初始值都是 system clock + option(2T) ，而 P-VOP 的 PTS 則是上一個 frame 的 PTS 加上 133468 uSec = (4*T)，至於 B-VOP 的 PTS 則為上一個 frame 的 PTS 減去 33367 uSec=(1*T) ，其中 T = 1/29.97 Sec 。

4.5.1.2 T(i) 函式計算



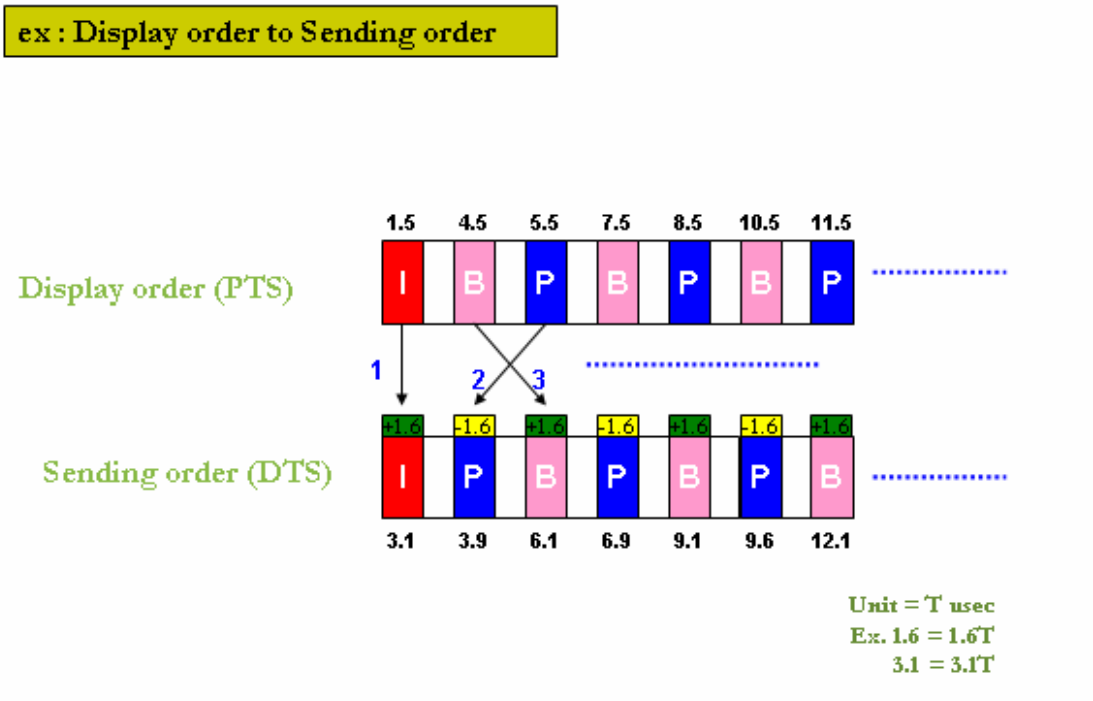


圖 4.8 T(i) 函式計算

由上圖 4.7 中我們運算出各個 frame 的 PTS 的值後，由於 PTS 只是為了計算

TimeStamp ， TimeStamp 是一種 Display order，以 Server 來講我們要的是傳送 order。 接下來我們要把 Display order 轉換成 Sending order，至於轉換的機制便是由圖 4.8 T(i) 函式運算後所得到的。如果是 I/B frame 時我們會將相對應 I/B frame 的 PTS 的值加上 1.6T，如果是 P frame 必須將相對應 P frame 的 PTS 減去 1.6T，進而得到 DTS 的值 。

4.5.1.3 TimeStamp 用途

TimeStamp 是一個在 RTP 表頭檔中的其中一個欄位的值，如下圖 4.7 所示，其主要的用途是為了要給用戶端的播放器在接收到 RTP 封包後，可以讓 Client 的播放器知道在哪個時間點可將此封包中的內容播放出來。由於 TimeStamp 是由 PTS 運算而來，但是封包傳遞的順序卻是按照 DTS，因此會發生 sequence number 比較大的封包，卻會有一個比較小的 TimeStamp。

如果此時某封包因為網路的關係，未能依 Sequence number 的先後順序抵達用戶端，再加上抵達時間已經超過該封包的播放時間，播放器在讀完 RTP 表頭的 TimeStamp 欄位後，確定這是一個**『過時』**的影音封包，便將此封包略過不做處理。

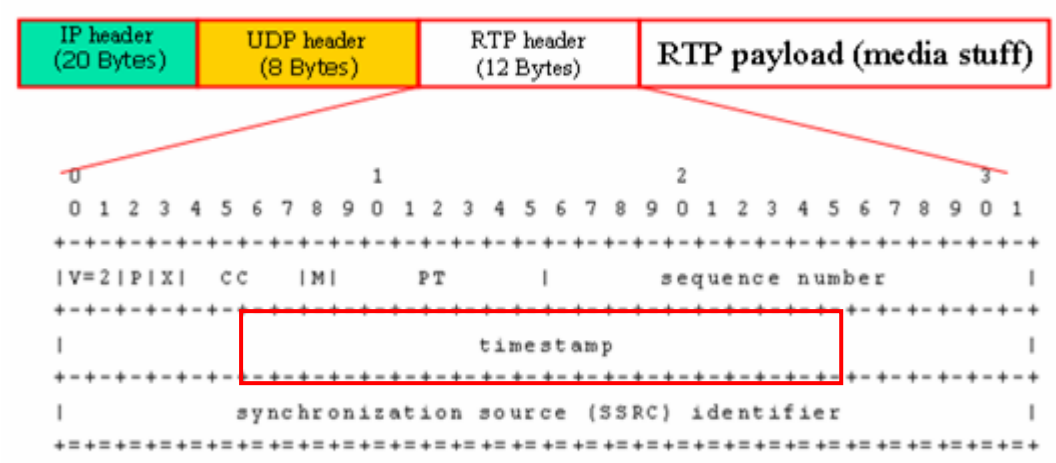


圖 4.9　RTP header structure

4.5.1.4　TimeStamp 計算方式

瞭解 TimeStamp 的重要性之後，接下來要說明的是 TimeStamp 是怎樣得來的。TimeStamp 是由 frame Display order (PTS)的值乘上 90kHz(90kHz frequency rate for Mpeg2)再加上一個亂數值得到的(計算公式如下所示)，因此 TimeStamp 必須透過 frame display order (PTS)的值來設定(PTS 的計算方式敘述於 4.5.1.3 小節)。

＊ TimeStamp = PTS * 90k +　一個亂數起始值 ***(In case of MPEG2)***

正因為 PTS 的時間順序便是影片播放的順序(IBPBP…，參考圖 4.2)，因此在經過這一轉換機制後，程式便可以『**循序**』設定 frame 的播放時間，也就是設定每個 frame 所屬封包的 TimeStamp；同時搭配 Sequence Number 後，串流用戶端的播放器便可以先由 Sequence Number 來排定封包播放的順序，其後再依

TimeStamp 於預計的播放時間點將封包內容解碼後播放之。

4.5.1.4 TimeStamp 後續處理

設定完每個 RTP 封包的 TimeStamp 後，我們可以說已經完成了 RTP 封包的『封裝』過程，也就是我們已經把 payload 加上 RTP header 而成為一個 RTP packet。但是我們如果直接將這樣的 PTS-frame(IBPBP…)順序經由切割封裝送出去，用戶端的播放器將無法解碼。

無法解碼的原因是伺服器傳給播放器的 frame 順序必須是 DTS-frame 的順序，因為解碼需要的順序是 IPBPB…(P-frame 需要 I-frame 的資訊，B-frame 需要前/後一個 I/P-frame 的資訊才得以解碼)，因此我們有必要將 PTS-frame 的順序再還原為原來 DTS-frame 的順序，細部的時間 Reverse 過程便不再續述。

4.6 多用戶的訊框處理

截至目前為止，我們所討論的都是單一用戶的 frame 切割封裝流程，接下來要討論的是伺服器的多用戶 frame 處理流程。首先我們來看下圖 4.10 的多用戶訊框處理示意圖。

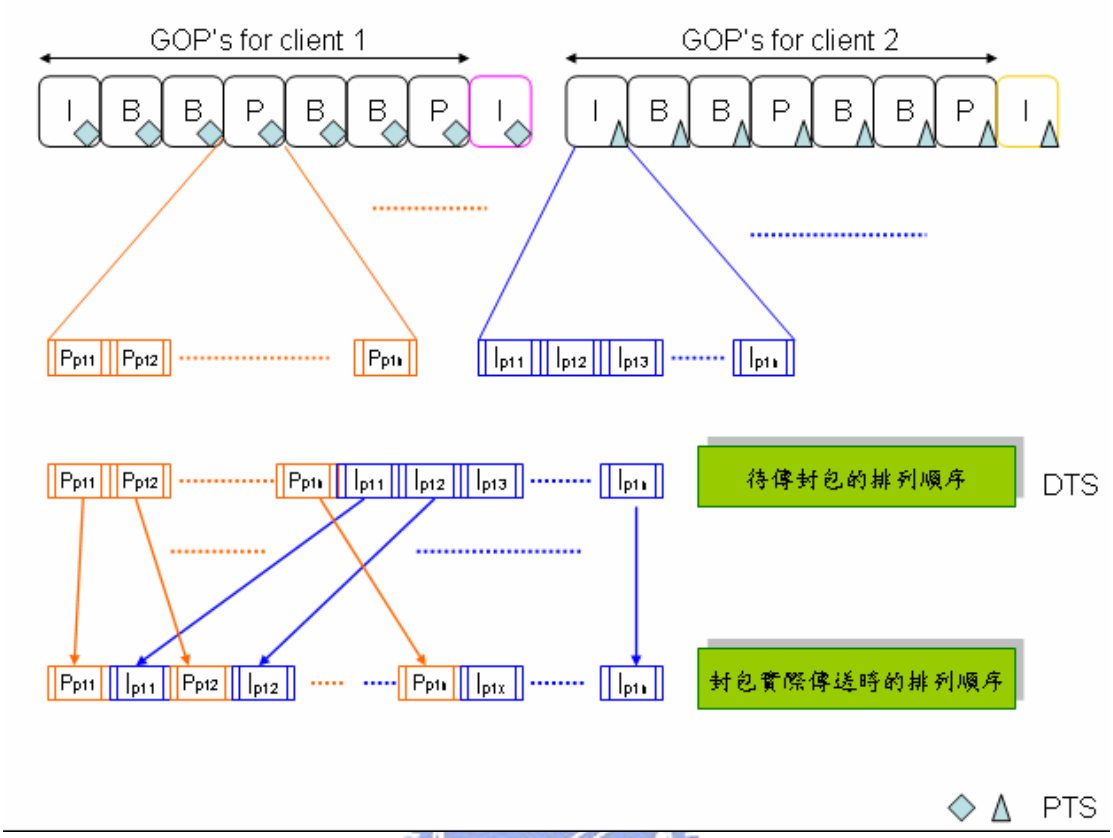


圖 4. 10 多用戶訊框處理示意

在圖 4. 10 中，伺服器以訊框(frame)為一個單位『**循序**』地處理 **client 1 以及 client** 2 的訊框，經由切割之後成為圖 4.10 下方的數個封包，而這些封包內容(packet payload)會先儲存在伺服器的**輸出封包緩衝區(fOutPacketBuf)** 中，直到伺服器認為『**時機成熟**』才把這些封包內容給傳送出去，其背後的含意是為了要做到『**收送速率的匹配**』，因為 Tx(傳送端)傳送速度如果大於 Rx(接收端)的接收速度，則 Rx 的 Buffer 將會 explosive；反之 Rx 大於 Tx 的話，便是播放器沒有 media payload 可以播，串流用戶便只能看著播放器的黑色畫面發呆。

而在呼叫 BSD Socket API 把封包內容給傳送出去前，我們還必須要完成『封裝』的動作，也就是把 RTP header 加在封包內容之前。而這些貼上 TimeStamp 的封包最後便是以 DTS-frame 的順序將已完成切割封裝的 RTP 封包經由呼叫 BSD Socket API 傳送到網路上，而在 OSI 的第四層 Transport layer 中，作業系統核心會自動替 RTP packet 加上 UDP header；第三層 Network layer 則會替 UDP packet 加上 IP header，成為一個可以在網路上傳送的標準結構化封包。

因此，伺服器便需要有一套判斷『**時機成熟**』與否的機制，用來判斷緩衝區的封包內容是否該被傳送出去，同時這個時間控制(Scheduling mechanism)機制也要兼顧多用戶間的『**公平性**』，因為每個串流用戶的收視權利都是相同的。而這個機制也是我們 4.6.1 章節要繼續論述的。

4.6.1 多用戶封包的傳送時間控制

在單一用戶連線情況下，Iterative Server 只要不斷的去處理該用戶所要求的 frame，經過切割封裝後便可以將封包給傳送出去，雖然**單用戶**與**多用戶**的封包**傳送時間控制**很像，但我們在此要強調的是伺服器封包多工的 Scheduling mechanism，其**設計理念**才是我們要探索的核心所在。

接下來我們實際以 Live555 來進行解說：

首先我們解釋以下兩個變數的用途 ~

1. fNextSendTime ：代表下一個**封包**要傳送出去的時間基準點(TimeBase)。

2. TimeNow(Wall clock) ：代表目前系統的時間 (以上的單位都是微秒) 。

在 4.6 章節中，我們提到完成切割封裝的封包都存放在伺服器的輸出封包緩衝區中(fOutPacketbuf)中，而同一個 VOP frame 所屬封包的 fNextSendTime 值都是一樣的，fNextSendTime 的設定方法如下所示：

(1). 1st_frame_fNextSendTime：即是設定為系統的目前時間(Wall Clock)，這是為了能夠馬上回復用戶端的需求，因此伺服器會立刻把 1st_frame 經過切割封裝處理後的封包傳送出去，第一個 frame 的 fNextSendTime 設定方式如下程式碼所示：

fNextSendTime = Wall Clock + delay time ;

(2). 2nd_frame_fNextSendTime：當呼叫 BSD socket API 把 1st_frame 所屬封包傳送出去後，我們便設定同一個用戶的下一個 frame 所屬封包的 fNextSendTime，設定下一個 frame 的方式為：

fNextSendTime += durationInMicroseconds ;

(其中的 durationInMicroseconds 便是 T(1/29.97 sec))

(3). 第二個用戶其 frame 所屬封包的 fNextSendTime 設定方式同(1)，因為 Wall Clock(壁鐘，也就是系統目前的時間)是一直持續增加的，因此如下圖 4.11(a) 所示，當 TimeNow 的 usec 欄位大於 Client1/2 的 fNextSendTime 的 usec 欄位，則伺服器便將此 payload 送出去做切割封裝後傳送出去給使用者。

(PS. TimeNow(Wall clock)是系統目前的時間，fNextSendTime 則是 I/P/B frame 切割時便指定封包會傳送的時間，因此同一個 frame 所切割出來的封包會具有同樣的 fNextSendTime(封包傳送時間)) 。傳送封包時，伺服器如何做到公平性是接下來的重點。為了顧及每個使用者有觀看影片的公平性，伺服器必須以 round robin 的方式來傳送封包，至於如何以 round robin 的方式傳送封包，這部分我們可分成為以下 A、B、C 來說明。並請參考下圖 4. 11(a)、4. 11(b) ：

(A)： 如紅框 A 所示，當 TimeNow 未大於 C21(Client2) 的 fNextSendTime 時，行程便專注於傳送 Client1 的封包(C11、C12、C13) 。Client2 的封包傳送時間尚未到達，所以伺服器暫時無法傳送封包給 Client 2。

(B)：如紅框 B 所示，當行程發現 Client2 TimeNow 大於 fNextSendTime 的話，行程便採用 Round Robin(輪循式)的方式交錯傳送 Client1 以及 Client2 的封包(C21、C14、C22、C15、C23、C16、……)。

(C)：如紅框 C 所示，當行程發現 Client1 的封包已經傳送完之後，行程便專注傳送 Client2 的封包(C2x、C2x、C2x，……)。

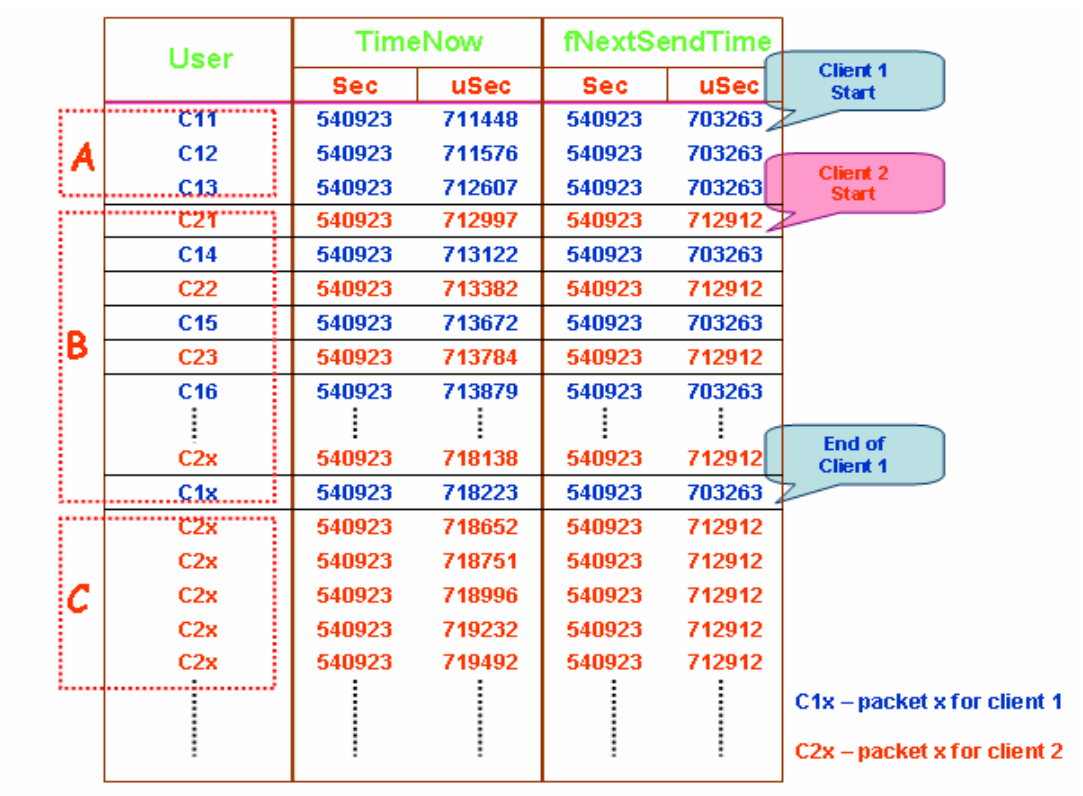


圖 4. 11(a) 用戶行程切換示意圖

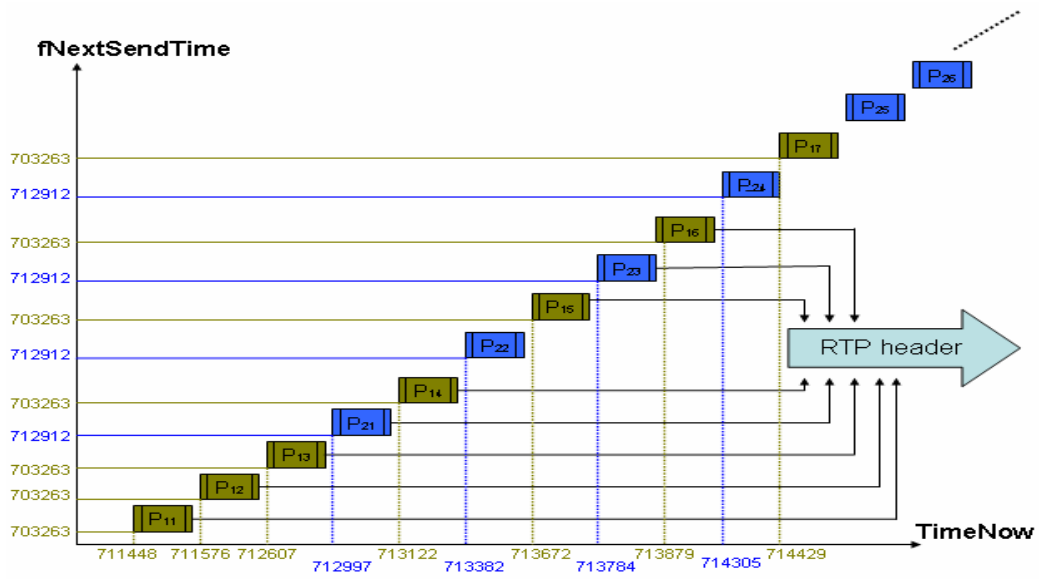


圖 4.11　(b) 用戶行程切換示意圖

如下圖 4. 12、4.13 所示，伺服器切換服務三個用戶的的時間數據分析，其封包傳送的演算法如前所述，故在此不再贅述。

User	TimeNow		fNextSendTime	
	Sec	uSec	Sec	uSec
C11	393092	262039	393092	261459
C12	393092	262545	393092	261459
C13	393092	262806	393092	261459
C21	393092	263029	393092	262595
C14	393092	263475	393092	261459
C22	393092	263628	393092	262595
C15	393092	263969	393092	294825
C23	393092	264207	393092	262595
C24	393092	264633	393092	295961
C1x	393092	298810	393092	294825
C2x	393092	302368	393092	295961
C1x	393092	302902	393092	294825
C31	393092	303078	393092	303007
C2x	393092	303198	393092	295961
C1x	393092	303673	393092	294825
⋮	⋮	⋮	⋮	⋮
C2x	393092	310817	393092	295961
C32	393092	333045	393092	303007

Client 1 Start
Client 2 Start
TimeNow Duration 34058
Client 3 Start
TimeNow Duration 22228

圖 4. 12　三個用戶連線的時間數據分析(續下圖 4.13)

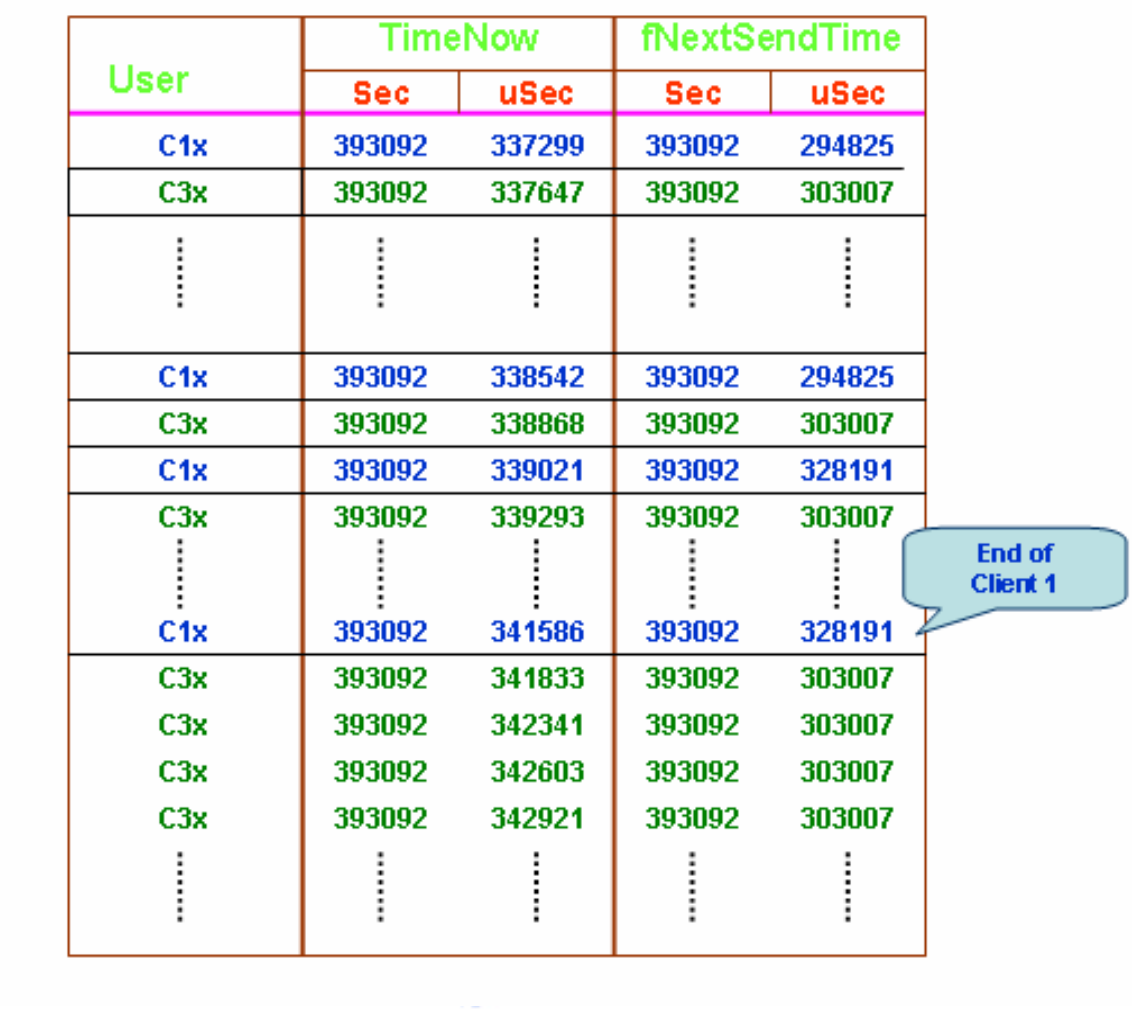


圖 4.13　三個用戶連線的時間數據分析(續上圖 4.12)

第五章　結論

Streaming Server 任務中包含了 RTSP 信令解析、媒體影音檔案格式的判斷、封包切割封裝以及傳送的時間控制，其中封包的傳送時間控制在伺服器的多工系統中是相得值得研究的議題。

如我們在第四章中提到的，在 Client-Server 架構的應用中，最大的問題往往是收送速率匹配的問題，而此問題衍生出來的研究便是去探討如何有效因應網路頻寬的動態來調整 Server 的傳輸速率，也就是當目前網路頻寬使用率不高時，Server 可以動態提高傳輸速率以增加 QoS 的品質；反之當目前網路頻寬使用率很高(網路擁塞)時，Server 可以降低傳輸速率，但仍須維持基本 QoS 的品質。

而在我們目前的研究中尚未考慮到頻寬使用率的問題，Server 在傳送封包時只是按照一個固定的時間(如每 1/29.97 秒)將所有要傳送的封包傳至用戶端，一旦發生網路擁塞時也許在 1/29.97 秒內，封包未能傳達到用戶端，導致用戶端發生很嚴重的間隔或連線中斷。例如手機在通話的過程中，如果 Channel 不好將會使得通話品質受到影響，或者收訊時好時壞，但以上的情形都是用戶端最不想看到的情形。

以串流伺服器的應用來說,很重要的考量因素便是如何提供一個穩定的影像/音訊品質，而一個好的設計便是在有限的資源下做出效益最大的取捨，也就是我們必須在 QoS 以及軟硬體資源中做出取捨。舉例來說，Buffer 可以提供用戶穩定的影音品質，假設在 1/29.97 秒內 Server 沒有傳送封包給用戶端，用戶端仍可在 1/29.97 秒內繼續播放影音不受影響，故在 Client 端考慮的重點將會是 Buffer 的設計與配置問題。

回到 Server 來說，在這 1/29.97 秒中可以繼續去做 frame 的切割封裝的動作，並考慮 Buffer 的大小來決定是否繼續進行切割封裝，而這個 Scheduling Mechanism 的機制是我們可以去控制的，也就是要確保在 BandWidth/Channel 不佳的情況下，Server 如何維持穩定的多工連線品質(QoS)，這也是本論文的研究目地。

而後 Home/Traffic Surveillance/Office System 的應用將無所不在，其產品如 IPCam、STB(Set Top Box)以及 VOIP 相關的嵌入式產品其背後都需要有**『小而美』**的軟體來維持系統的穩定運作，而應用面不同其多工排程演算法(Scheduling Mechanism for Multiplexing)便需要適應環境而變，如何提供一個更精緻的**多工排程演算法**將會是本論文以後的研究方向。

參考文獻

[1] http://ffmpeg.mplayerhq.hu/

[2] http://www.live555.com/

[3] http://www.videolan.org/vlc/

[4] http://www.faqs.org/rfcs/rfc2616.html RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1

[5] http://www.youtube.com

[6] http://www.nuuo.com

[7] http://www.mingw.org/download.shtml

[8] http://www.libsdl.org/download-1.2.php

[9] H.Schulzrinne , A.Rao, and R.Lanphier, "Real Time Streaming Protocol(RTSP)", RFC2326,April 1998

[10] H. Schulzrinne, Columbia University, “RTP：A Transport Protocol for Real-Time ” ，RFC3550. Titl , July2003

[11] E.COMER , L.STEVENS , “INTERNETWORKING with TCP/IP” , Volume 3 Publisher ：Prentice Hall

[12] 陳重嘉，experience in software development concepts，disciplines and cases

[13] 吳宗修，Analysis of Streaming Server’s Properties，國立交通大學，96學年度

[14] UNIX網路程式設計，網路應用程式設計介面Socket與XTI
作者：W. Richard Stevens ，譯者：林慶德，出版：培生

[15] 多媒體通訊，原理・標準・與系統第二版
作者：戴顯權、陳瀅如、王春清編著，出版：紳藍

[16] Silberschatz , Galvin and Gagne , “Operating System Principles”, SIXTH EDITION , Publisher ：WILEY

[17] William Stallings , Operation Systems：Internals and Design Principles, 5th ed. , Publisher：Prentice Hall , 2004

附錄 一：如何在 windows 下去編譯以及執行 ffserver

Step[1] 連上 http://www.mingw.org/download.shtml 網頁，下載 MinGW-5.0.2 及 MSYSmys-1.0.10 套件。

Step[2] 先安裝 MinGW-5.0.2，之後再將解壓縮後的 MSYSmsys-1.0.10 安裝在 MinGW-5.0.2 裡的 bin 資料夾內。

Step[3] 連上 http://ffmpeg.mplayerhq.hu/download.html 網頁，下載 Ffmpeg 的 source code。

Step[4] 開啟 Msys 的執行檔，進入 Ffmpeg 的目錄下鍵入

```
./configure --enable-memalign-hack
make
make install
```

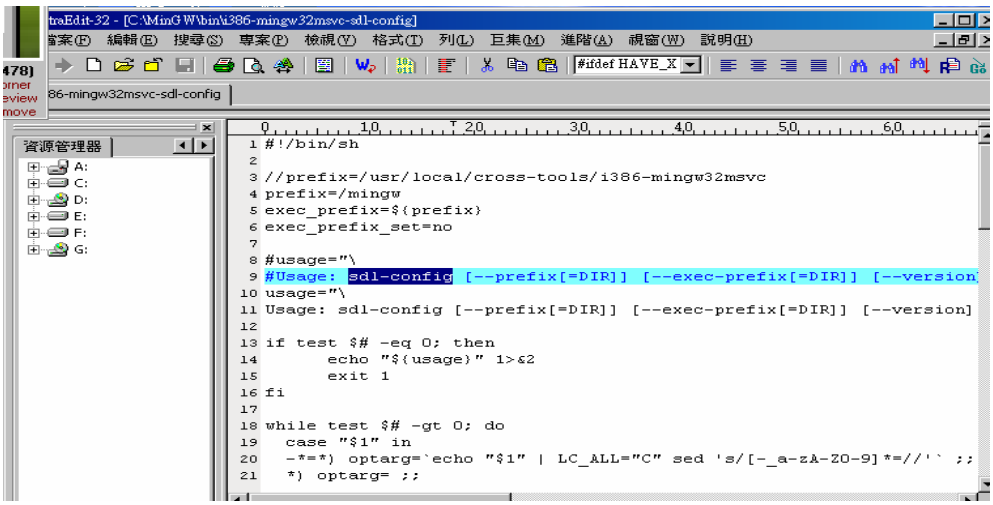
Setp[5] 根據上面的步驟可以成功的編譯出 ffmpeg.exe 檔 與 ffserver.exe 檔。

Step[6] 想要編譯出 ffplay.exe 檔,必須要安裝 SDL(Simple DirectMedia Layer) 從此 http://www.libsdl.org/download-1.2.php 網頁中下載。

Setp[7] For (Mingw32) SDL-devel-1.2.11-mingw32.tar.gz 版本下載的網頁，下載 SDL 後將它解壓，之後會看到 bin , include , lib , share,四個資料夾，將這四個資料夾 copy 到 MinGW 的目錄下，把原來的檔復蓋過去。之後在 Window 底下使用 MinGW 去編譯。

Step[8] 進入 MinGW 的 bin 目錄下修改 i386-mingw32msvc-sdl-config 文件的第一行為 Prefix=/mingw。 之後再將 i386-mingw32msvc-sdl-config 檔名改為 sdl-config。

因為 i386-mingw32msvc-sdl-config 的文件中發現 usage(資料來源)是在 sdl-config，所以一定要將 i386-mingw32msvc-sdl-config 檔名改為 sdl-config。否則編譯會出錯。



Step[9] 開啟 Msys 的執行檔，進入 Msys ffmpeg 目錄底下鍵入

./configure --enable-memalign-hack

Make

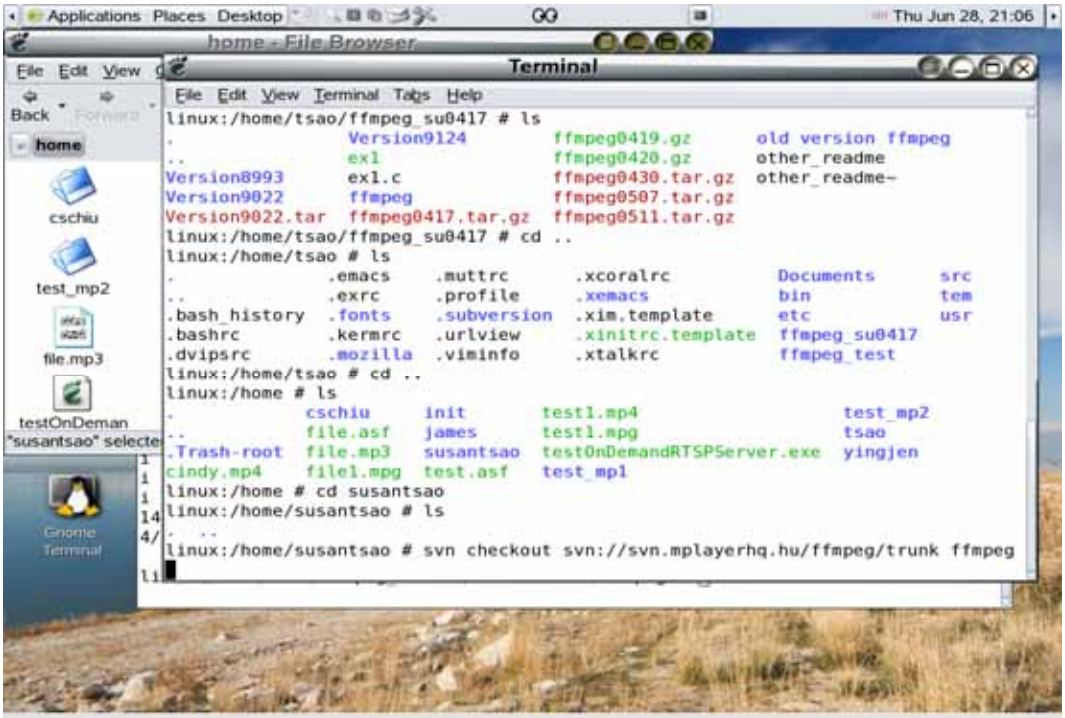
Make install

Setp[10] 即完成編譯的動作。此時在 ffmpeg 的目錄下會看到 ffmpeg.exe，ffserver.exe 與及 ffplay.exe 檔。

附錄 二： 在 Suse –linux 作業系統下編譯與及執行 ffserver

Step 1：在 Suse-linux 作業系統上，使用 SVN 下載 ffmpeg

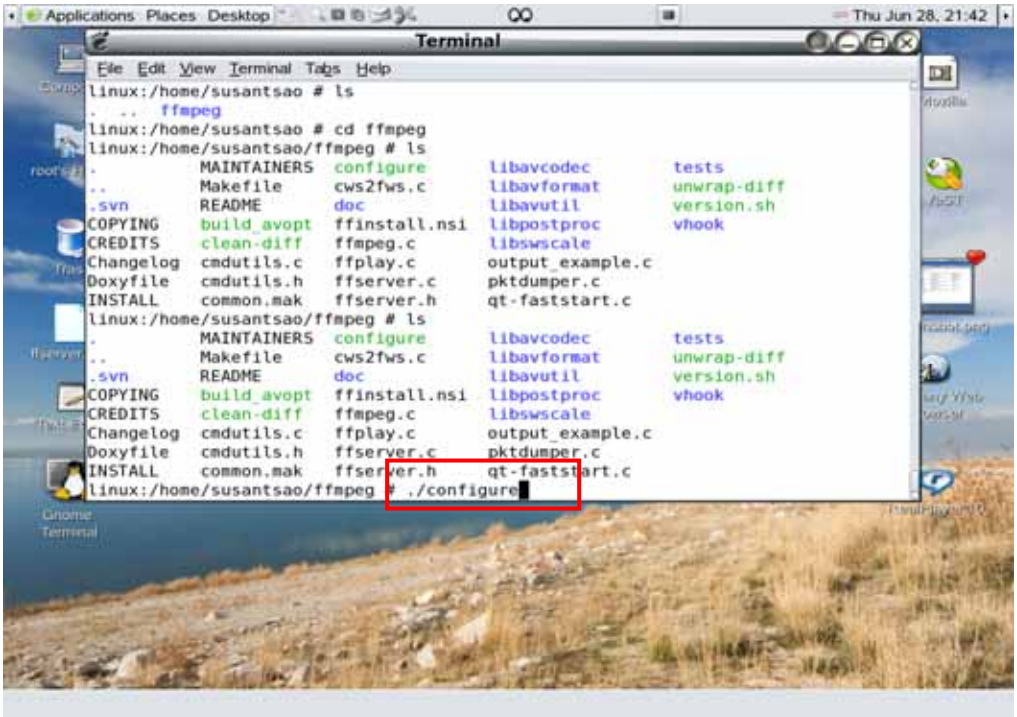
(svn checkout svn://svn.mplayerhq.hu/ffmpeg/trunk ffmpeg)



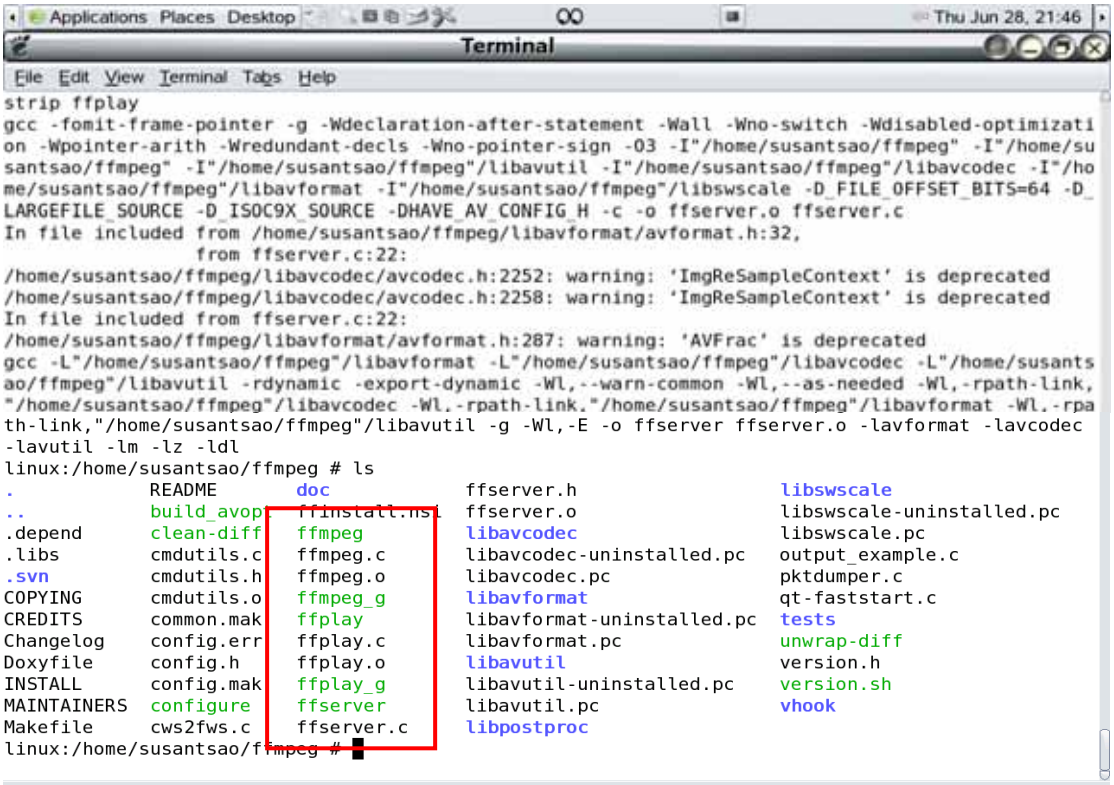
Step 2：在 ffmpeg 目錄下去執行 config 與 make 的動作

./configure

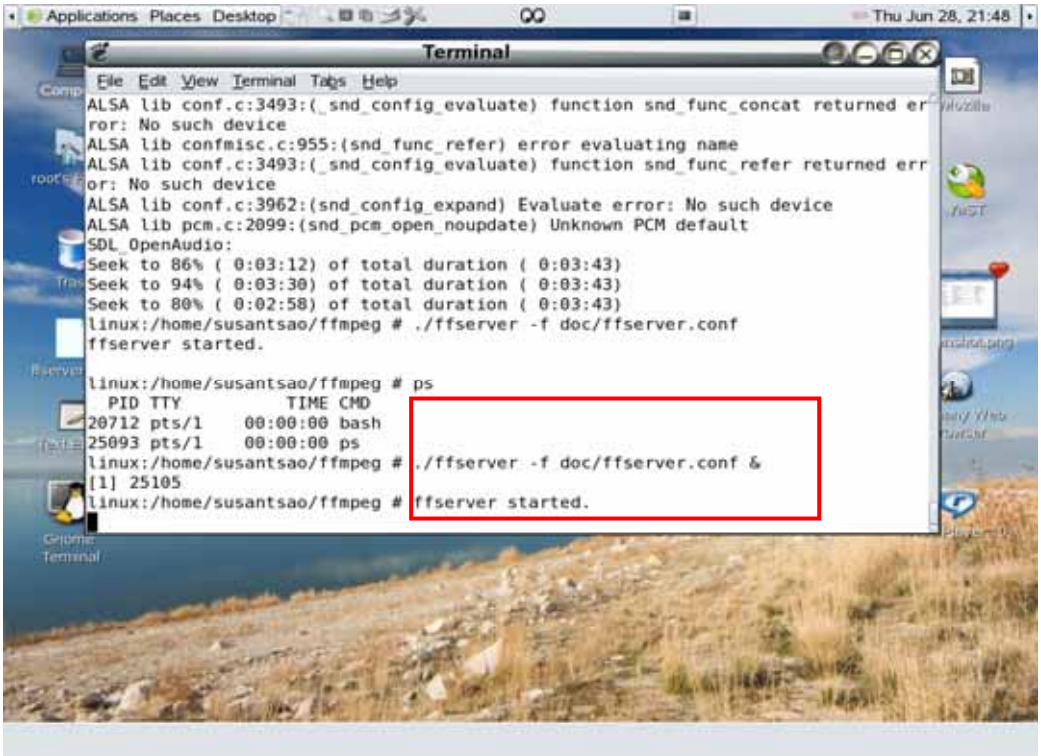
./make



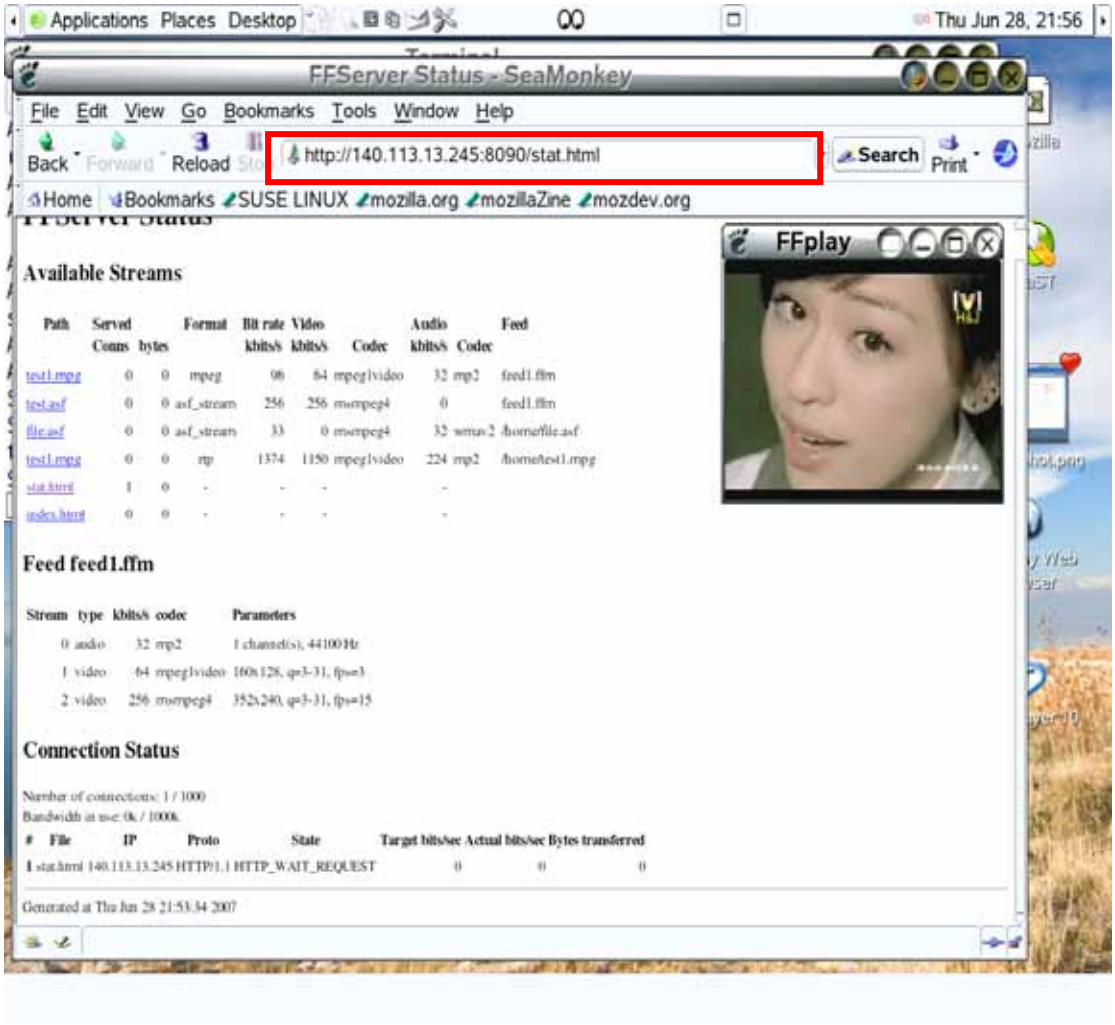
Step 3：執行 make 動作之會，我們在 ffmpeg 目錄下可看到多出 3 個執行檔 Ffmpeg，ffserver，以及 ffplay，如下紅框內綠色字所示



Step 4：在 ffmpeg 目錄下鍵入./ffserver –f doc/ffserver.conf & 即可看到 ffserver Started ，表示已將 ffserver 伺服器已開啟



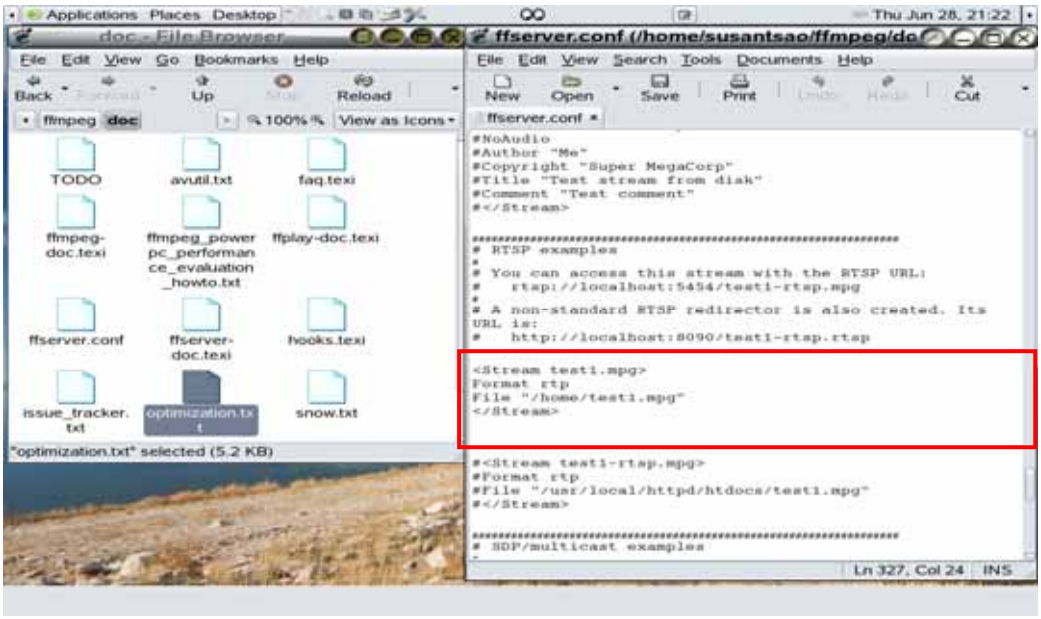
Step 5：使用 IE，鍵入 http://destination ip:port/stat.html，連接到 ffserver，即可看 Ffserver 預設的網頁圖，如下圖所示



[註：ffserver 的 HTTP 的預設 port 為 8090]

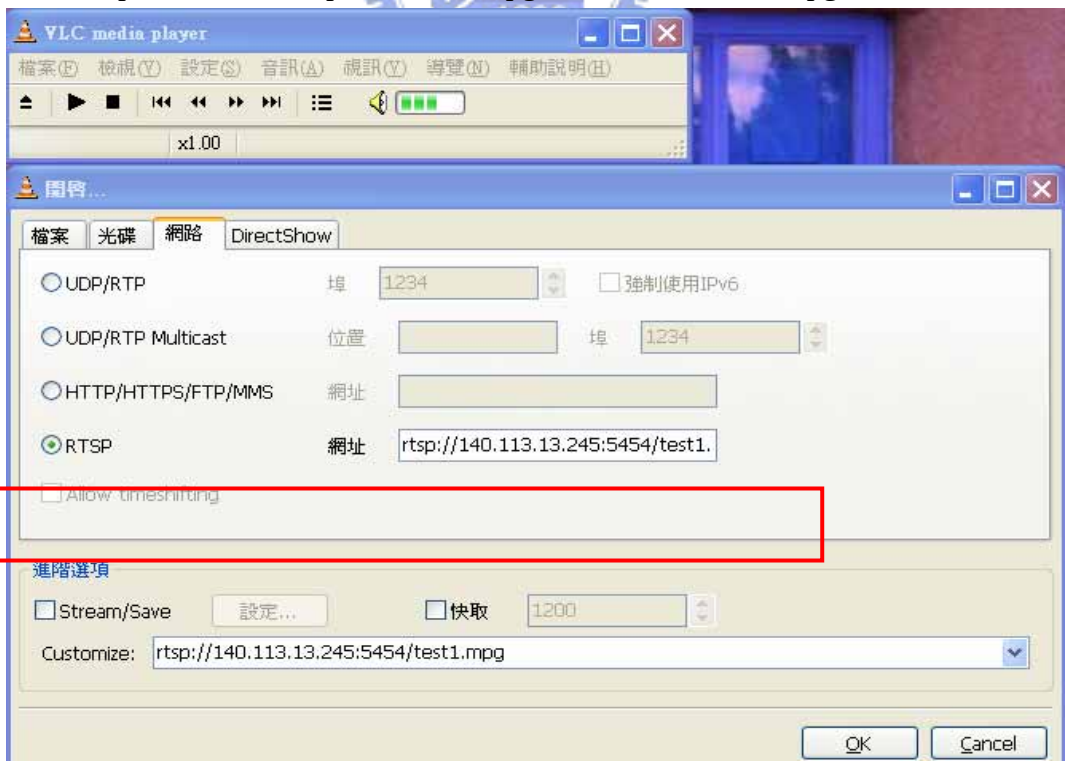
Step 6：接下來使用 VLC media player 連接到 ffserver

先進入 doc/ffserver.conf 檔中，將想要播放的檔案與路徑設定好，如下紅框所示



Step2：使用 VLC 播放器，在檔案->開啟網路串流->選擇 RTSP 在網址中鍵入

rtsp:// destination ip.Port/test1.mpg，即可看到 test1.mpg 播放的影片，

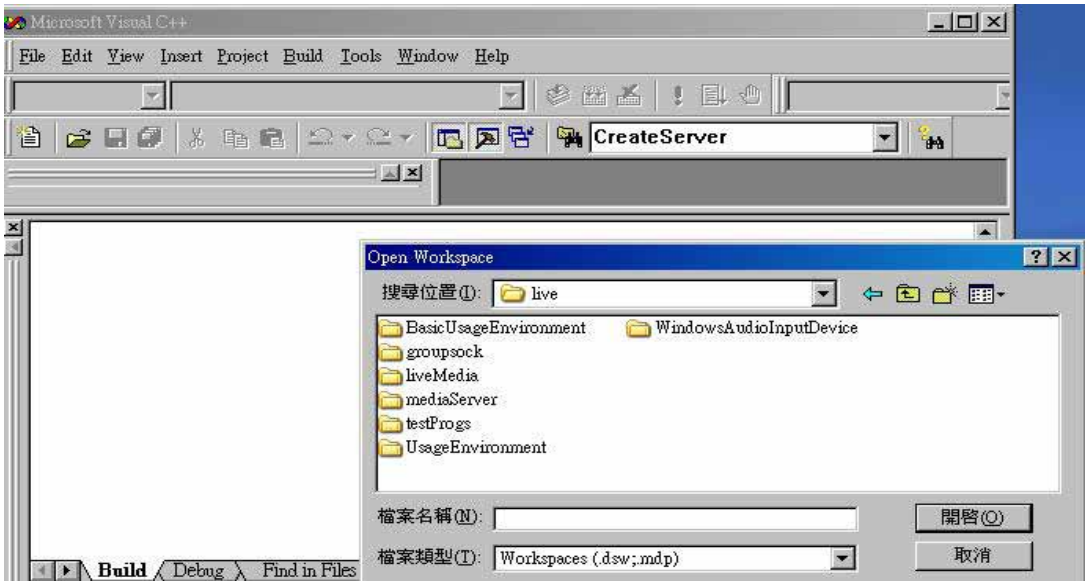


附錄 三：如何在 Windows 下去編譯以及執行 live555

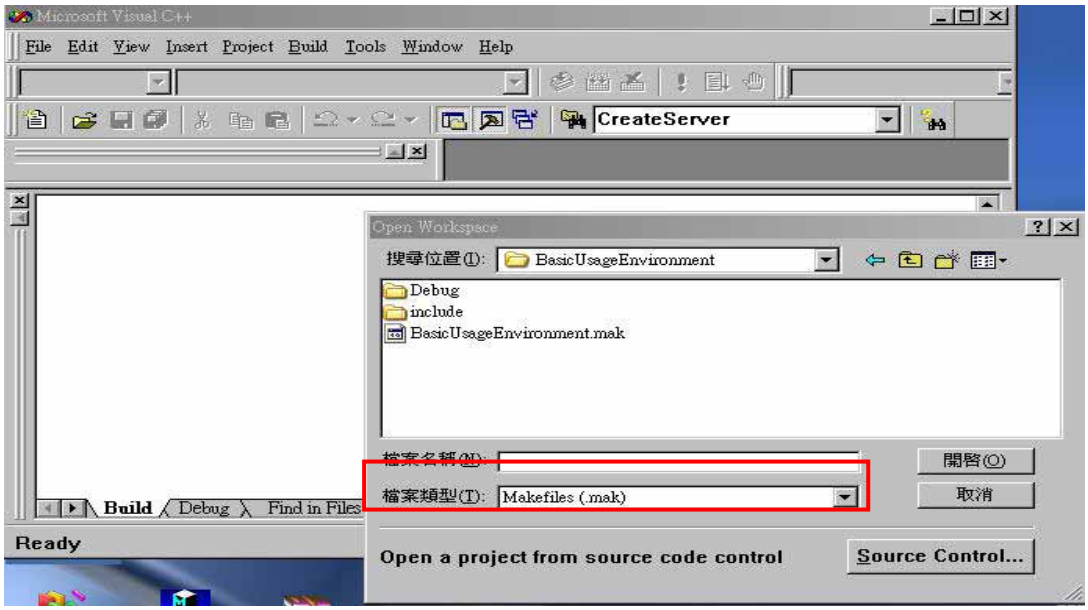
Step 1：先到此網頁 http://www.live555.com/下載 live555 的 Source code

Step 2：在 live555 目錄下有一個 win32config 檔案，將這檔案開啟後，在 Tools32 的路徑，更改為目前 VC++的正確的路徑。

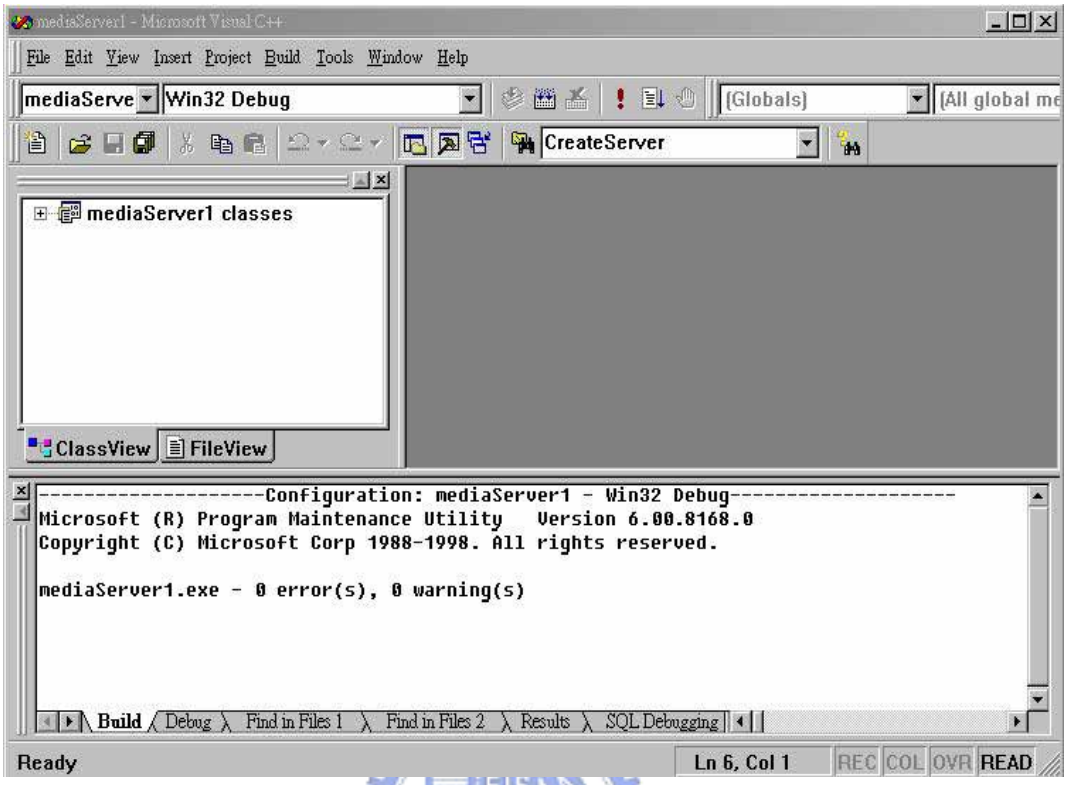
Step 3：打開 VC++ ，在 File ->Open Workspace ->找到存放 live555 的位置，在 live 目錄下會看到很多個目錄，如 BasicUsageEnvironment , testProgs…. 如下圖所示



Step 4：在檔案類別的地方，選擇副檔名為 Makefiles(.mak)檔

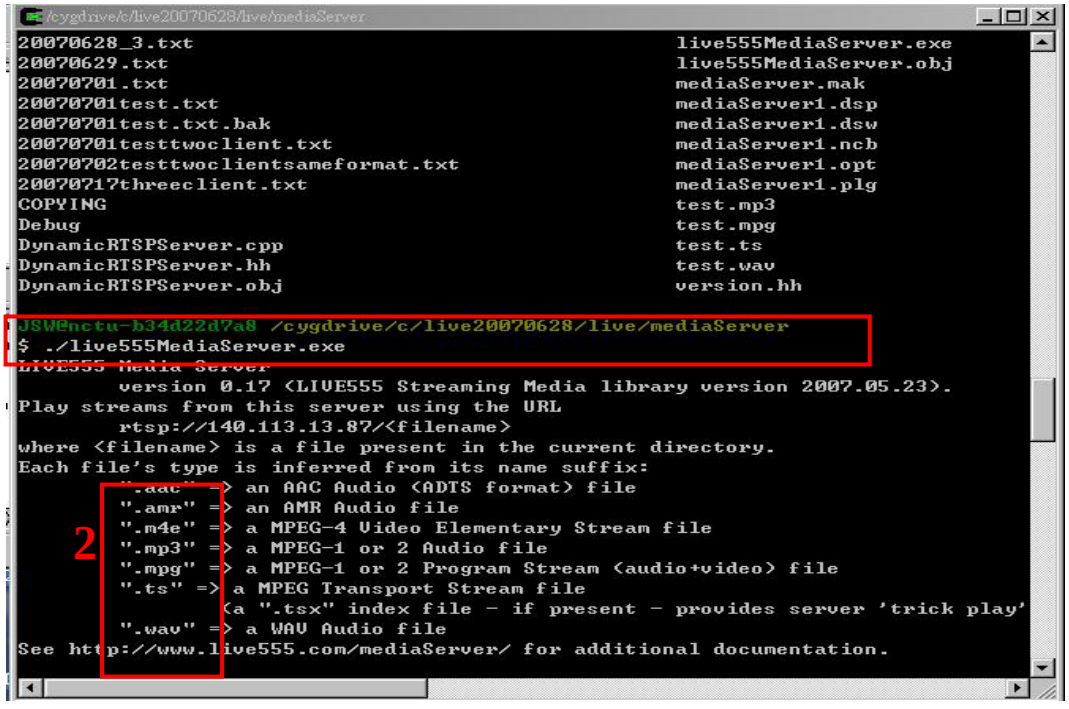


Step 5：接下來開始 build 檔案。

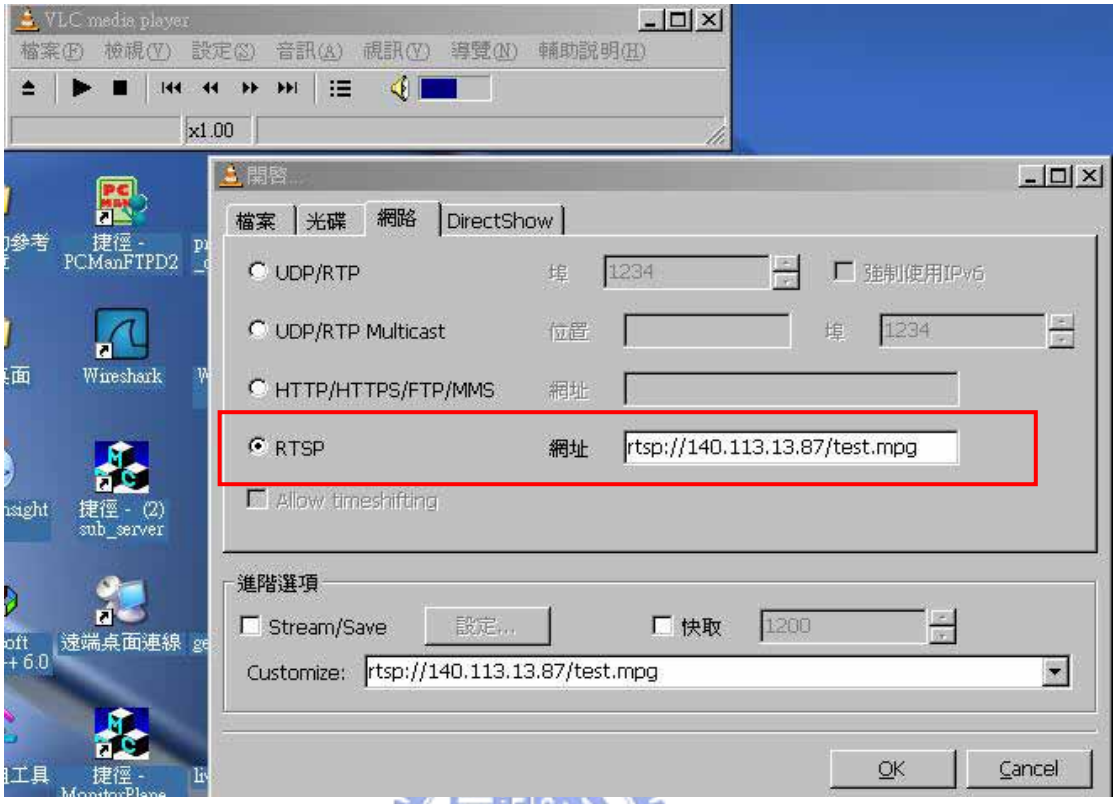


Step 6：重復 Step 3 ~ Step 5 直到將 live555 目錄中所有的資料都 build 完。

Step 7：接下來使用 cywin 將 live555 伺服器打開，我們看到在 live555 中預設可播放的影片如下圖 2 所示



Step 8：使用 VLC media player 連到 live555 伺服器，即可觀賞 test.mpg 的影片。

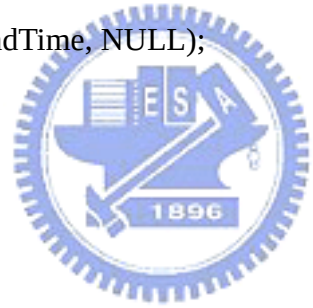


附錄四：Trace 程式時所使用的到的程式碼流程

1. 計算 fNextSendTime 的程式碼在 MultiFramedRTPSink.cpp 裡

A. void MultiFramedRTPSink

```
::afterGettingFrame1(unsigned frameSize, unsigned numTruncatedBytes,
                     struct timeval presentationTime,
                     unsigned durationInMicroseconds) {
  if (fIsFirstPacket) {
    // Record the fact that we're starting to play now:
    gettimeofday(&fNextSendTime, NULL);
  }
```



B. afterGettingFrame1()內

```
//Update the time at which the next packet should be sent, based
// on the duration of the frame that we just packed into it.
// However, if this frame has overflow data remaining, then don't
// count its duration yet.
    if (overflowBytes == 0) {
      fNextSendTime.tv_usec += durationInMicroseconds;
      fNextSendTime.tv_sec += fNextSendTime.tv_usec/1000000;
```

```
        fNextSendTime.tv_usec %= 1000000;
    }
```

2. 計算 PTS 的程式碼在-MPEGVideoStreamFramer.cpp 裡

```
void MPEGVideoStreamFramer
::computePresentationTime(unsigned numAdditionalPictures)
  // Computes "fPresentationTime" from the most recent GOP's
  // time_code, along with the "numAdditionalPictures" parameter:
 TimeCode& tc = fCurGOPTimeCode;

  double pictureTime = fFrameRate == 0.0 ? 0.0
    : (tc.pictures + fPicturesAdjustment + numAdditionalPictures)/fFrameRate
    - fPictureTimeBase;

  unsigned pictureSeconds = (unsigned)pictureTime;
  double pictureFractionOfSecond = pictureTime - (double)pictureSeconds;
```

```
//    fTcSecsBase = (((tc.days*24)+tc.hours)*60+tc.minutes)*60+tc.seconds;
    unsigned tcSecs
        = (((tc.days*24)+tc.hours)*60+tc.minutes)*60+tc.seconds - fTcSecsBase;
    fPresentationTime = fPresentationTimeBase;
    fPresentationTime.tv_sec += tcSecs + pictureSeconds;
    fPresentationTime.tv_usec += (long)(pictureFractionOfSecond*1000000.0);
    printf("**EnterComputePresentationTime****\n");
    printf("fpresentationTime.tv_usec=%d\n",fPresentationTime.tv_usec);
    printf("**End**\n");
    if (fPresentationTime.tv_usec >= 1000000) {
        fPresentationTime.tv_usec -= 1000000;
        ++fPresentationTime.tv_sec;
    }
```

3. 在建立 RTP 環境時的程式碼在 RTPSink.cpp 裡

A. RTPSink::RTPSink(UsageEnvironment& env,

```
        Groupsock* rtpGS, unsigned char rtpPayloadType,
        unsigned rtpTimestampFrequency,
```

```
        char const* rtpPayloadFormatName,
        unsigned numChannels)
  : MediaSink(env), fRTPInterface(this, rtpGS),
    fRTPPayloadType(rtpPayloadType),
    fPacketCount(0), fOctetCount(0), fTotalOctetCount(0),
    fTimestampFrequency(rtpTimestampFrequency),
    fNumChannels(numChannels) {
  fRTPPayloadFormatName
    = strDup(rtpPayloadFormatName == NULL ? "???" : rtpPayloadFormatName);
  gettimeofday(&fCreationTime, NULL);
  fTotalOctetCountStartTime = fCreationTime;

  fSeqNo = (unsigned short)our_random();
  fSSRC = (unsigned)our_random();
  fTimestampBase = (unsigned)our_random();
  fCurrentTimestamp = fTimestampBase;

  fTransmissionStatsDB = new RTPTransmissionStatsDB(*this);
}
```

B. 計算 Timestamp

convertToRTPTimestamp()函數裡：

```
u_int32_t RTPSink::convertToRTPTimestamp(struct timeval tv, Boolean isFirstTime)
{
    if (isFirstTime) {
      // Make the first timestamp the same as the current "fTimestampBase", so
that
      // timestamps begin with the value we promised when this "RTPSink" was
created:
      u_int32_t rtpTimestampIncrement = timevalToTimestamp(tv);
      fTimestampBase -= rtpTimestampIncrement;
    }
    return convertToRTPTimestamp(tv);
  }
```

timevalToTimestamp() 函數：

```
u_int32_t RTPSink::timevalToTimestamp(struct timeval tv) const {
//pts*90k+一個亂數基本值
u_int32_t timestamp = (fTimestampFrequency*tv.tv_sec);
```

```
timestamp += (unsigned)((2.0*fTimestampFrequency*tv.tv_usec + 1000000.0)/2000000);
                // note: rounding
    return timestamp;
}
```