國 立 交 通 大 學

電信工程學系

碩 士 論 文

多媒體通訊系統中之即時平順演算法

Real-time smoothing algorithm for multi-media communication system

研究生　：　吳心賢

指導教授　：　張文鐘　博士

中　華　民　國　九十四　年　十　月

多媒體通訊系統中之即時平順演算法

Real-time smoothing algorithm for multi-media communication system

研 究 生：吳心賢　　Student：Sin-Sian Wu

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學

電 信 工 程 系

碩 士 論 文

A Thesis

Submitted to Department of Communication Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Communication Engineering

October 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年十月

多媒體通訊系統中之即時平順演算法

研究生: 吳心賢　指導教授: 張文鐘 博士
國立交通大學電信工程學系碩士班

中文摘要

隨著頻寬的增加與視訊壓縮技術的進步和電腦網路技術的提升, 透過網路觀賞高品質即視訊服務的應用大大增多, 網路必須分送不同的影像資料以滿足不同的應用. 例如, 包含聲音的即時影像要求、多媒體短片與電影。這類資料會使用變動位元速率的視訊壓縮方法, 這種壓縮方式雖然可以保有每張圖片品質的一致, 但是在傳送時會有速率突然增加的情況(burst), 這會加重網路的負擔, 同時降低了網路的使用效率, 也使得網路的管理更加困難。平順演順法可以將 burst 變成一個平緩的資料流, 對整個網路的負擔會減少很多. 本論文主要是延伸已有的 stored video 平順演算法, 原本只能平順傳送已經存成檔案的影音資訊外, 現在也能夠處理即時產生的資料. 我們一開始先回顧基本的平順演算法, 說明其運作方式及特性, 接著再探討即時資料和已存好的資料在平順處理上有何差異, 針對這些不同之處改進原有的演算法. 並且實際將演算法實作在一個傳輸系統上, 原始的平順演算法必須因應實作的需求和限制而有修改, 才能相容於以封包為單位的傳輸系統. 最後我們對系統所使用的無線網路做了關於延遲及

jitter 等的量測,並從 MAC 的層面解釋網路對平順演算法的結果有哪些影響.

Real-time smoothing algorithm for multi-media communication system

Student: Sin-Sian Wu
Advisor: Dr. Wen-Thong Chang
The Department of Communication Engineering
National Chiao Tung University

Abstract

With rapid increases in network bandwidth and major advances in video compression technology, the number of the applications that provide consumers with high quality audiovisual contents has gone up dramatically. Hence, the network now has to support the transmissions of different types of data, such as real-time conferences, multimedia clips, and Hollywood movies. These different types of data, however, do have a common characteristic, “VBR”, which results from modern compression techniques to ensure that the quality of the contents maintains the same throughout the entire presentation. Since the bitrate of VBR contents is not constant, the traffic generated when transmitting them will obviously be bursty. This burstiness will add burdens to the network, decrease the utilization efficiency of the network resources, and complicate the task to manage the network. To solve this problem, the smoothing technique that can smooth bursty traffic in effect and hence remove some of the burdens caused by VBR contents has been proposed and developed.

In this work, the stored video smoothing technique that originally can only generate a smoother plan for stored audiovisual contents is extended. The proposed online smoothing technique can deal with real-time captured data with only minor buffering. In the beginning of the paper, the general smoothing techniques are reviewed, and their mechanisms and characteristics are illustrated. Subsequently, the major differences that stored and real-time captured contents can make to the smoothing algorithms are discussed. With the discoveries of these differences, how the original smoothing technique is changed so it can adapt to the online case is presented. Moreover, this extended algorithm is further modified so it can be implemented in a real transmission system. Specifically, since in the real world, the data are transmitted in packets but not in sequential bytes, the smoothing algorithm has to be changed to packet-based. Finally, after the technique is implemented in

the real system, measurements of delays and jitters in the wireless environment are taken. Their effects to the smoothing technique are explained in terms of the mechanisms of MAC layer.

誌　謝

碩士生涯， 終於有驚無險地完成了。過程中有甘有苦，感謝我的家人，在這段時間完全放手的讓我去尋找自己的道路。感謝指導教授張文鐘教授，口試委員范國清教授、黃仲陵教授和李安謙教授，你們的指導，讓我的論文更趨完整。

實驗室一起走過的朋友們，從明昇，承軒…到呆鵝, 11 號, 壞人瑋, 旃偉, 智偉, 其瑩; 再到為棟, 鮪魚哲, 小棟棟, 小鉉鉉…真是繁蔟不及備載啊(汗). 這段歲月, 因為有你們的相伴而增色不少. 一起討論問題, 一起打球, 吃飯, 唱歌, 聯誼的時光, 在將來一定是回味無窮的記憶.

其實我也很想有個女朋友可以感謝, 無奈天生好人命, 身邊都只有女性友人. 不過還是很感謝小呆, 圓圓, Q 靜, 室友大姐, 蝦球, 小薰, 花花…無聊時陪我聊天打發時間.

除了感謝身旁的朋友, 還要感謝上天讓我生在這個文化富饒的時代, 有機會可以學習人類千百年來累積的智慧. 要感謝的東西太多了, 所以…就謝天吧!

目　錄

表　目　錄

圖　目　錄

第一章 緒論

1.1 研究動機

隨著頻寬的增加與視訊壓縮技術的進步和電腦網路技術的提升，透過網路觀賞高品質即視訊服務的應用大大增多，整合性應用的服務也增多了，網路必須分送不同的影像資料以滿足不同的應用，例如，包含聲音的即時影像要求、多媒體短片與電影。為了能提供更高品質的服務，我們要克服一些問題。 影像串流被認為未來網路影用的主要部份。其中一個重要的問題，變動位元速率的視訊壓縮方式雖然可以保有每張圖片品質的一致，但是在傳送時會有速率突然增加的情況(burst)， 這類本身含有大量突衝以及速率變動的資料流，會加重網路的負擔，同時降低了網路的使用效率，也使得網路的管理更加困難。如果可以把它變成一個平緩的資料流，對整個網路的負擔會減少很多，本篇論文介紹的平順演順法就是要做這件事。

然而，除了理論之外，我們還希望能將 smoothing 演算法實作在一個實際的即時傳輸統上，所以要克服許多實務上的困難，並對基本的演算法做出修改。主要的修正是為了讓平順演算法能應用在以封包為單位傳送的實際傳輸系統上，並且可以對事先存好的影片(stored

video)或是即時傳送的(online)資料做 smoothing。

此篇論文主要是延讀[13]和[15]的討論，針對他們所提出的 smoothing 演算法做改進。

1.2 論文架構

論文的第一章是緒論，介紹研究動機和基本背景，第二章介紹 smoothing 演算法的概念及運作流程。第三章進一步探討如何在傳送即時影像時做 smoothing，第四章展示在 MATLAB 上的模擬結果，並比較此演算法和未修改前的效能差異。第五章是敘述如何實作在一個實際的傳輸系統。第六章是整篇論文的結論。

第二章　平順演算法

2.1 平順演算法簡介

本篇論文的主題是 online smoothing，在更深入探討怎麼做 online smoothing 之前，我們先談談什麼是"平順演算法"(smoothing algorithm)。在傳輸資料的時候，例如說即時傳送一部影片，會發現傳輸這個影片檔案時， 每秒鐘的傳輸量有大有小，傳輸速率並不是固定的;在場景變化很多的片段會有較大的資料要傳，此時會需要較高的頻寬，而在碰到靜態的內心戲橋段時，傳輸量又小很多，相較於一般時候是浪費頻寬。如果為了滿足影片最高傳輸速率而給他這樣高的頻寬，那麼在其它時間，都會是浪費頻寬的情況。

smoothing 就是為了將忽大忽小的傳輸速率轉變成很平穩的傳輸速率，甚至是固定速率(CBR， Constant bit rate)產生的技術。使用了 smoothing 的技術，可以幫助傳送端更容易地管理傳輸速率，進而提升能夠負荷的使用人數上限。

CBR， variable quality 壓縮的視訊，複雜性較低，但使用者看起來的品質也較差; VBR(variable bit rate)， constant quality 壓出來的視訊品質就好很多，但時它的資料量會隨時間而有很大的變

化。以現今常用的數位視訊壓縮技術 MPEG 來說，短期的資料暴衝(burst)是因為 I，P，B frame 包含的資料不同所致，長期的資料變化則是因為 scene content 不同的關係。簡單來說，smoothing 基本上就是為了克服 VBR， constant quality 的視訊壓縮所造成的資料量忽然爆增的缺點。(圖)是示意 smoothing 排程造成的改變。

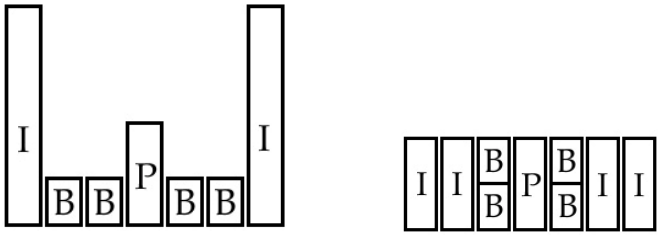


圖 2.1　(a)未經過 smoothing 前　　(b)smoothing 後的排程

接下來的問題是，我們怎麼做才能達到 smoothing 的效果?最簡單且直覺的方法就是預先傳送(work-ahead)。首先在接收端安置一個緩衝區(buffer)，然後在接收端開始播放影片之前預先將資料平穩地傳入接收端的 buffer 內，在等待了一小段時間的延遲後(數秒~數十秒，看其應用和使用者能接受的程度)， 接收端才開始播放影片。因為預先屯積了一些"糧食"，所以遇到播放端播出要"吃"掉較多資料的畫面時，仍不會出問題。而且傳輸速率可以維持在一個平穩的狀態。簡單地說，smoothing 主要是算出一個排程，安排傳送端在何時要丟出多少量的資料。

我們接著介紹一下，現在有哪些 smoothing 演算法。為了讓算出

的排程中，傳輸速率改變次數最小，W. Feng and S. Sechrest 提出[3]的 MCBA(Minimum changes bandwidth allocation);而為了讓算出的排程中，傳輸速率的標準差最小(整體的傳輸速率最集中)，J. D. Salehi ，D. Towsley 等人設計出[1]的 MVBA(Minimum variability bandwidth allocation)，而 W. Feng and S. Sechrest 提出[5]的 CBA 則是為了最小化排程中速率增加的次數。其中我們以 MVBA 為本篇論文的談論核心。

為何要選 MVBA?因為實際的情況下，接收端的 buffer 大小是定死的(硬體已經做好了)，而 MVBA 演算法也是假設一個固定的 buffer 大小來運算。而且本篇論文要實作的 online smoothing 是架構在一個已經實作完成的 stored video smoothing 演算法上，而此 smoothing 演算法是 MVBA，所以我們選 MVBA 當作找排程的方法。此章接下來的部份，是說明 smoothing 演算法的運作流程。

2.2 傳輸排程上下限

首先，我們先定出傳輸排程的上限及下限:傳輸的下限是，接收端播放時要吃掉的資料量。如果傳送端沒有在接收端播放這張影片之前，餵足夠的資料給接收端，就會發生 underflow 的情況，也就是接收端會播不出畫面，或是畫面中有某些區塊會花掉，不論何種情形，

都是使用者不能容忍的。

我們設d_i為第 i 個單位時間接收端播第 i 個 frame 需要的資料量，(本篇論文中皆以一個 frame 時間長度當作一個單位時間)簡言之，d_i就是影片中第 i 個 frame 的大小。另外，為了演算法的說明方便，我們設$D(t)$為從開始到 t 為止，所累積的d_i:

$$D(t)=\sum_{i=0}^{t} d_i \ , \qquad \text{for all } 0<t<N \qquad (2\text{-}1)$$

$D(t)$就是時間 t 所需累積的資料量下限。

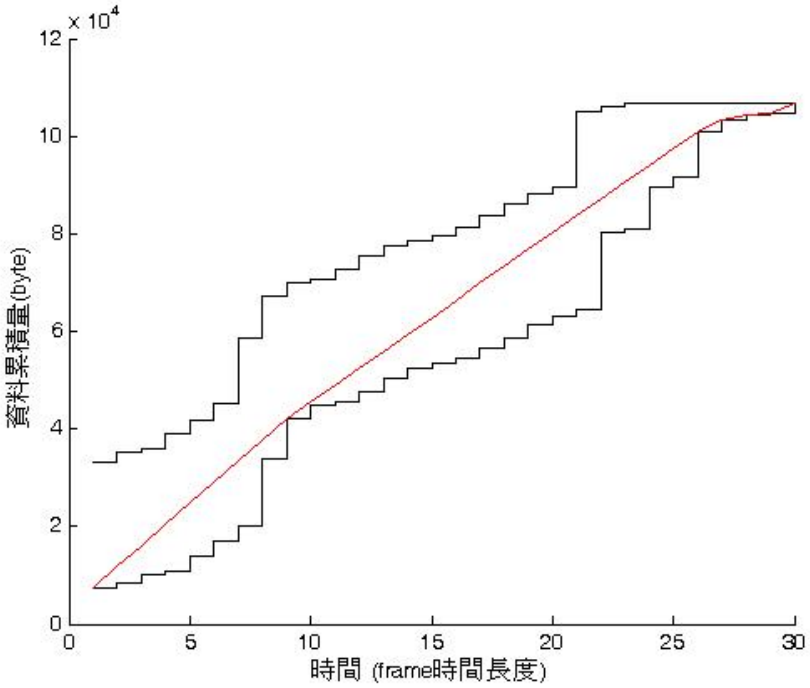
再來，傳輸上限是最低傳輸量再加上 buffer 大小，我們令傳輸上限$B(t)$為:

$$B(t)=D(t)+b \ , \qquad (2\text{-}2)$$

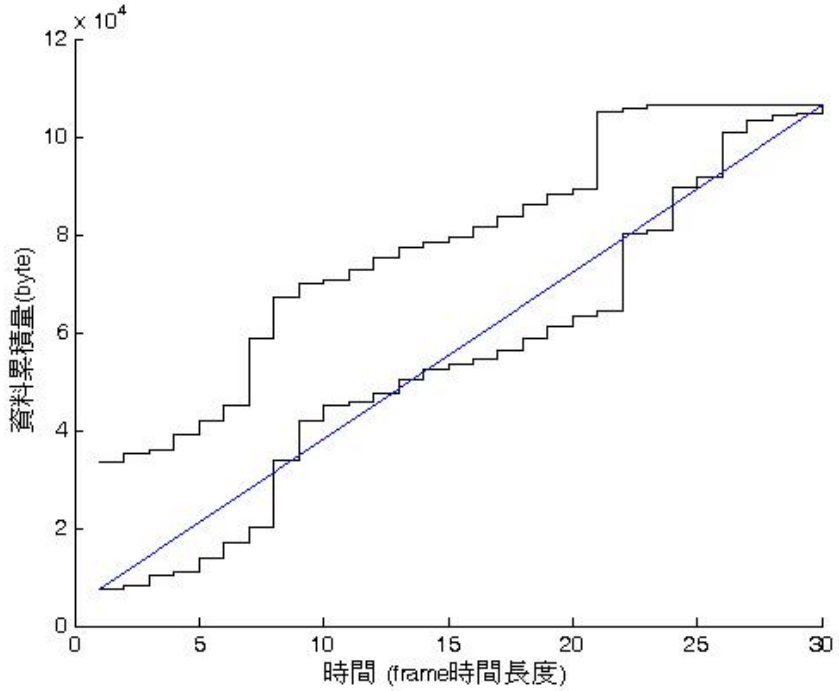
我們的傳輸排程必須要在上限與下限之間才行，也就是排程所規劃傳送出去的資料累積量$\sum_{i=0}^{t} s_i$必須滿足下式:

$$D(t)\le\sum_{i=0}^{t} s_i \le B(t) \ , \qquad \text{for all } 0<t<N \qquad (2\text{-}3)$$

$B(t)$和$D(t)$各是一條曲線，我們要找一條線，在上升的過程中不能超過$B(t)$或是低於$D(t)$。例如說圖 2.2 就是個可接受的排程，而圖 2.3 就是個不能接受的排程(發生 underflow)。



(圖 2.2)在上下限之間的排程



(圖 2.3)低於下限的排程(underflow)

2.3 MVBA 平順演算法

我們已經知道，要找一個在上下限之間的排程，但要怎麼找才是最好的呢?最理想的情況是能找出一個 CBR 的傳輸速率，但是實際上很難做到整個影片的長度只有一個 CBR 來傳;所以我們採用 piecewise CBR 的方式，也就是由許多段 CBR 所組成的排程。選取的方式是以可以延伸最長的片段為主，一直到了會發生 underflow 或 overflow 的情況時，才做一次轉折(改變速率)。轉折點(Critical point)都會放在上限或是下限的角角處，在後面的圖例中可以觀察到。圖 2.4 是 MVBA 的 block diagram 圖。

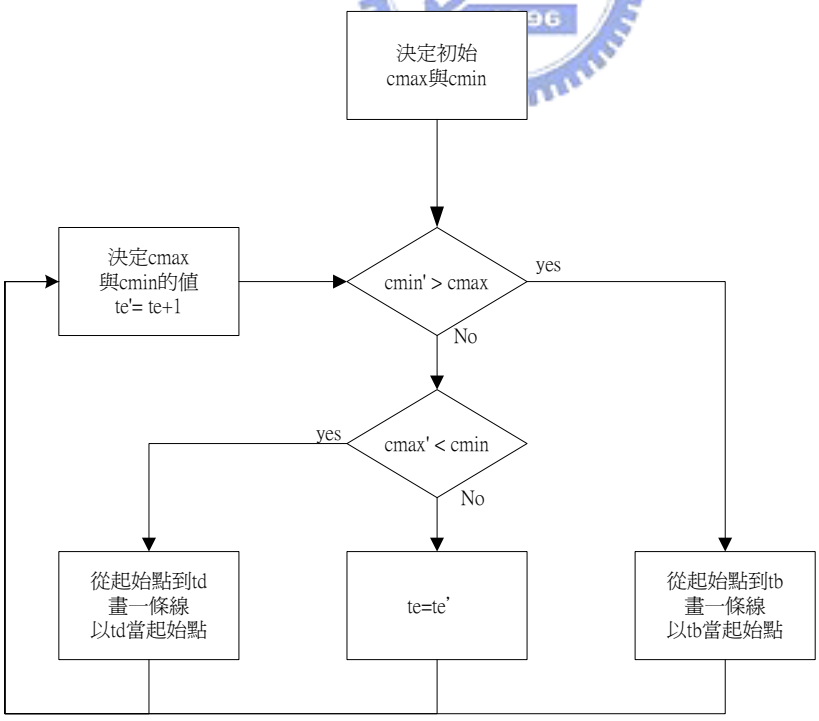


圖 2.4 MVBA 流程圖

MVBA 的 pseudo-code 列出如下:

```
//set inital values
1  ts = 0, te =1, t_B =1, t_D =1
2  while (ts ≤ N)
3  {
     //advance one index and compute feasible rate
4     te' = te +1
5     if ( we can't find feasible rate, and the buffer will be underflow)
6     { output a segment(ts~t_B);let next ts = t_B, te=ts+1 }
7     else if ( we can't find feasible rate, and the buffer will be overflow or te'==N)
8     { output a segment(ts~t_D);let next ts = t_D, te=ts+1 }
9     else
10     {
11        compare and save the feasible rate C_max, C_min;
12        if C_max, C_min are updated, save the corresponding critical point(t_D, t_B);
13        te = te';
14     }
15 }
```

MVBA optimal smoothing algorithm pseudo-codes

接著我們敘述一下其運算過程。一開始我們以時間 0 為起點，然後先給定一些預設值，如 cmax=b; cmin=d(1)，td=tb=1;cmax 代表從起點到 te 為止，可以合法選擇的斜率中，最大的值。一般情況的計算方法如方程式(2-4)

$$c_{\max_{ts,te}} = \min_{ts+1 \le t \le te} \frac{B(t)-(D(ts)+q)}{t-ts} \quad , \qquad (2\text{-}4)$$

而 tb 是指得到這個 cmax 值的時間點;

$$t_{B_{ts,te}} = \max_{ts+1 \le t \le te} \left\{ t : \frac{B(t)-(D(ts)+q)}{t-ts} = c_{\max} \right\}. \qquad (2\text{-}5)$$

同理， cmin 代表從起點到 te 為止，不會超過上下限，最小的值。

$$C_{\min_{ts,te}} = \max_{ts+1 \le t \le te} \frac{B(t)-(D(ts)+q)}{t-ts} \ , \qquad (2\text{-}6)$$

而 td 是指得到這個 cmin 值的時間點;

$$t_{D_{ts,te}} = \max_{ts+1 \le t \le te} \left\{ t : \frac{D(t)-(B(ts)+q)}{t-ts} = C_{\min} \right\}. \qquad (2\text{-}7)$$

接著計算從起點 ts(初始值為 0)到時間 te’的最大和最小斜率。比較目前的 cmax 及 cmin，比較的結果有三種 case:

Case1: 起點到 te’的最小斜率大於目前的 cmax，表示之前合法能選的斜率不夠大，再往下就會 underflow 如圖 2.5。這種情況就定出一條 CBR，意即從起始點到 B(tb)畫條線，並選 tb 當起始點，重新開始。

圖 2.5 以 frame index164 為起點，當 te’前進到 index170 的時候就找不到可行的斜率了。因為從 ts 到 te’-1 的可行最小斜率 cmin 是在 index165 時發生，td=165，以藍色實線表示;cmax 在 index168 發生，tb=168，以紅色虛線表示。而從 ts 到 te’的最小斜率，即 B(164)到 D(170)連線的斜率，比紅色虛線還大，此即為 case1 的情況，所以我們定 B(tb)為轉折點，並定出一段 CBR(ts 到 tb 這段)。tb 之後

的下一段 CBR 是下一個迴圈決定的。

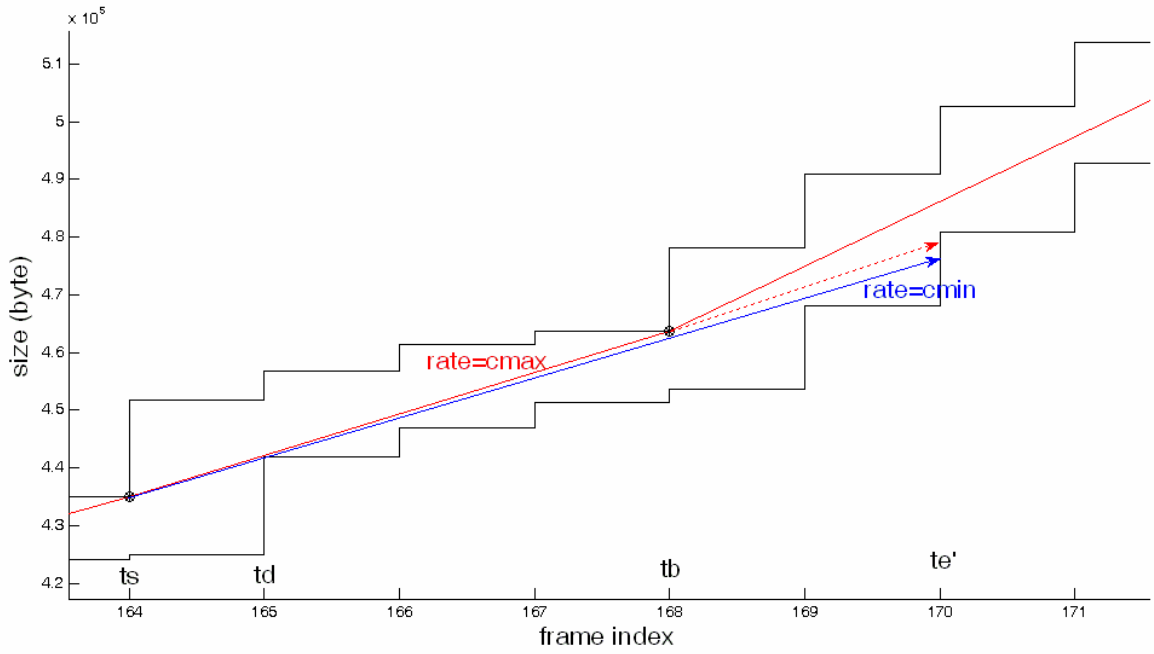


圖 2.5 不定轉折點就會 underflow 的情形

Case2: 起點到 te’ 的最大斜率小於目前的 cmin，表示之前合法能選的斜率不夠小，再往下就會 overflow 如圖 2.6。這種情況就定出一條 CBR，意即從起始點到 D(td)畫條線，並選 td 當起始點，重新開始。

圖 2.6 以 frame index 300 為起點，預設 td 和 tb 都是 index 301，預設 te’ 為 index 302，而此時就找不到可行的斜率了。因為從 ts 到 te’ 的最大斜率，即圖中黑色實線的斜率，比可行的最小斜率 cmin(藍色實線)還小;此即為 case2 的情況，所以我們定 D(td)為轉折點，並定出一段 CBR(ts 到 td 這段)。

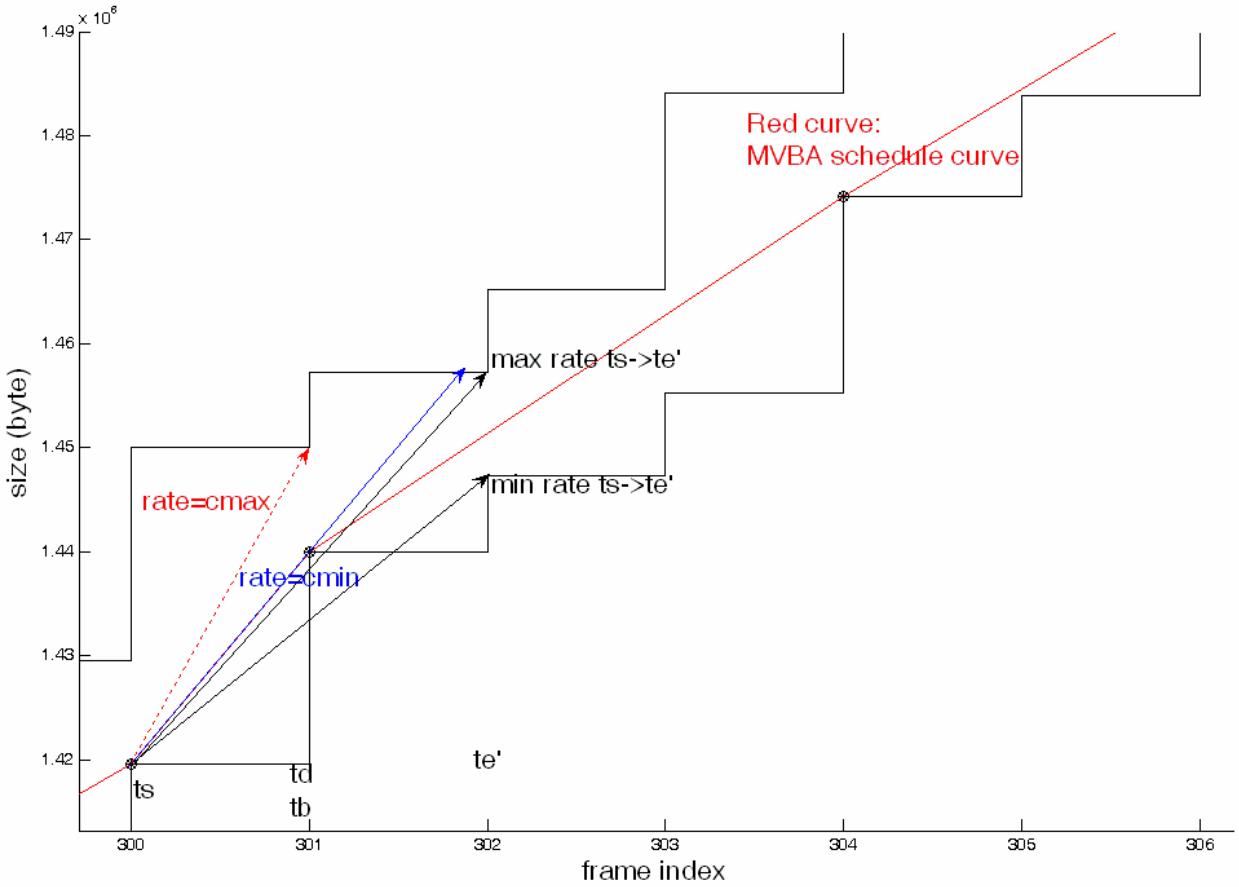


圖 2.6 不定轉折點就會 overflow 的情形

Case3:不是 case1 或 case2 的情況。我們令起點到 te’ 的最大/最小斜率為 cmax’/cmin’，比較 cmin 和 cmin’，取較大的當新的 cmin，並且令 td 等於較大那個斜率對應到的 t;同理，比較 cmax 和 cmax’，取小的當新的 cmax。並且令 tb 等於較大那個斜率對應到的 t，以圖 2.7 為例，ts=246，紅色虛線和藍色實線表示 te’=251 時的 cmax 和 cmin，而 tb=251，td=248。當 te’ 前進到 252 時，我們會維持紅色虛線的斜率為 cmax， tb 依舊是 251;綠色實線之斜率為新的 cmin，

而 td 由 248 更新成 252。最後令 te=te+1。

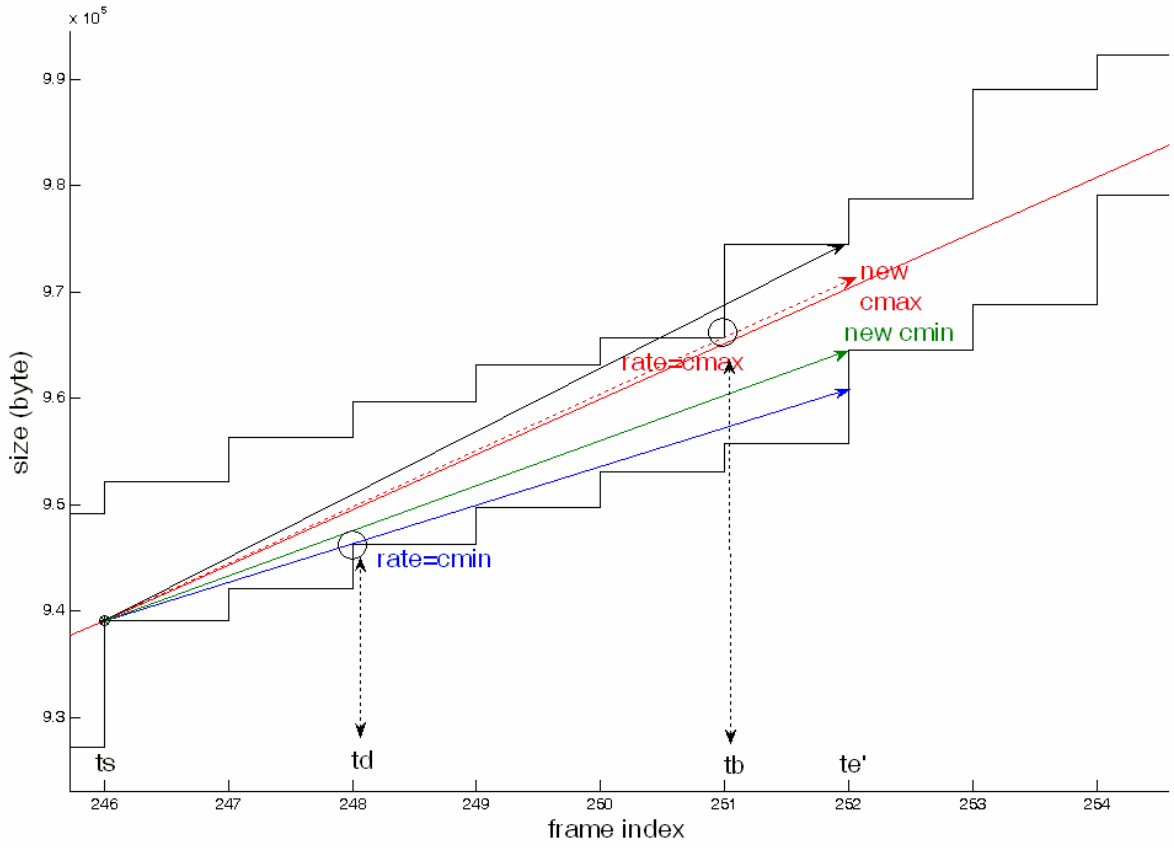


圖 2.7 case 3 的一個例子

下一次迴圈一開始會讓 te’=te+1，並重新計算從起始點 ts 到時間 te’的最大和最小斜率。然後再和目前的 cmax 及 cmin 比較，又會有三種情況…這個迴圈會一直持續到 ts == N 為止。

簡言之，演算法會想找出一條在上下限之間且能延伸最遠的線段，也就是 case 3 不斷往推進 te 的情況，一直到會有 overflow 或 underflow 才決定線段的終點(即 case 1 和 case 2 的狀況)，並以終點為新的起始點，再次重複延伸線段的動作，一直重複到整段影片結

束為止。

2.4 總結

最後對 smoothing 演算法的流程在此作一個總結，一開始根據接收端緩衝區的大小和播放的 video trace，依照 2-1 和 2-2 兩式建立上限及下限。然後找出一個在上下限之間最平順的排程。

MVBA 是找排程的演算法中的其中一種，特點是能保証有最小的 rate variance，也就是標準差會比其它演算法還小。若有興趣了解詳細的數學證明，可以參考[13]。

這個演算法原本是應用在 stored video 上的。我們將在下一章探討如何將此設計修改以到適用於 online 的情況。原本在 stored video 的情況下會考慮快轉和倒轉時該怎麼做 smoothing，但在 online 的情形下，即時播放的影像不會用到快轉和倒轉，所以這部份就不多做討論。

第三章 Online smoothing

3.1 Online 與 stored video

在前面一章已說明過 smoothing 的功用和作法， Smoothing 就是預先在有大流量之前先多傳一點資料.舉例來說，就像是已知未來一個月每天有多少工作量，為了減少在某段期間特別忙碌，而在較清閒的日子事先完成未來的工作.如此一來，可以避免工作量超過負荷的情況.且在想要同時做兩，三份工作的情況下，也方便分配與管理工作量.

Smoothing 要傳輸的資料來源可以分成 stored video (可以想成是 VOD， video on demand)和 online 兩種類型.STORED VIDEO 就是要傳送的資料都已經完整地放在傳送端的硬碟裡，像是電影檔案;而 online 類型則是指欲傳送的資料要即時產生，例如視訊會議，或是即時轉播的新聞，球賽.而同樣是 online 類型，又會因為使用者能夠容忍的延遲長短不同而需要不同的"即時性".例如說即時轉播就能有比視訊會議更大的延遲，看到延遲 10 秒鐘的球賽還沒什麼關係，但用視訊溝通時，10 秒鐘的延遲大概會讓人想砸電腦吧.

Smoothing 可說是一種排程的工作，然而針對上述兩種類型的資

料來源， smoothing 的做法也有不同. 我們先分析這兩類資料來源:

1. STORED VIDEO:因為我們傳送端已經知道所有要播放的資料，所以不管是每隔多久播出一個 frame，或是一個 frame 有多大，這些資料都是已知的.所以 smoothing 要做的事就是把這所有的資料，從頭到尾安排何時送資料，每次又送多少.
2. online:傳送資料即時產生，傳送端只知道數個 frame 時間長度的資料量，我們能做排程的內容就是傳送端現有的這些資料量.

瞭解 STORED VIDEO 和 online 有什麼異同之後，接下來的問題是:

1.online 的情況能否套用已有的演算法，也就是使用 STORED VIDEO 跑的 MVBA 演算法，如果可以套用，需不需要做什麼修改?

2.修改之後的 online MVBA，其效能比起 stored video 能一樣好嗎?抑或是較差?差了多少?

接著讓我們來回答上面兩個問題:第一個問題，會在接下來的 3-2 和 3-3 小節敘述修改的方法;第二個問題則放在第四章的模擬結果中討論.

3.2 fix size,non overlapping window smoothing

首先，我們已有一個用在 stored video 的 smoorhing 演算法，即 MVBA.若給它一段 N 個 frame 時間長度，每個 frame 各要傳 d(t)

個 bytes 大小的資料(1<t<N)，再給它接收端的 buffer 大小 B，此演算法就可以依此兩個參數，重新安排出這段 N 個 frame 的時間內，每個 frame 傳多少 bytes 的資料，使得整體傳輸速率的標準差最小.所以 MVBA 這個涵數的輸入有兩個: d(t)和 B，輸出是 A(t)， 1<t<N.

在 stored video 上，d(t)是已知的，因為整個電影檔都在手上，當然知道每秒要播出幾個 frame，每個 frame 又有多大，B 的大小就看接收端的 buffer 大小;MVBA 只要在開始傳送前，將整段 video 做一次運算，就能求得平順的傳輸流程; 如果是在 online 的情況，我們沒有完整的 d(t)可以當輸入，那怎麼辦呢?最簡單的做法，我們設定一個 window size=W，讓傳送端先等 W 個 frame 的時間，先在傳送端的 buffer 內累積 W 個 frame 的 d(t)，1<t<W，然後把這個 d(t)和 B 用 MVBA 的演算法去做 smoothing，就能求得接下來的 W 個 frame 時間的平順傳輸流程.傳送端每次知道了 W 個 frame 時間長度的資料後，就把這 W 個 frame 看成一個做 MVBA 的 window，每當傳送端多知道下一個 window 的 frame 資料時，就用 MVBA 算一次平順傳輸流程，決定未來這 W 個 frame 的時間內，每個 frame 要送多少資料.這種以 window 為單位的計算方法，我們稱為 fixed size， non-overlapping window smoothing，意即運算的範圍一次跳動一個 window.

在做 MATLAB 模擬的時候，我們可以把整個影片的 video trace

分割成很多個以 W 為一組的小段，以此模仿 online 情況下，傳送端收集一個 window(W 個 frame 時間長度)的動作，然後每個 window 做一次 MVBA.

這個做法的缺點是，如果 window 的第一個 frame 要傳大量資料，就無法預先傳送，如(圖 3.1)所示:

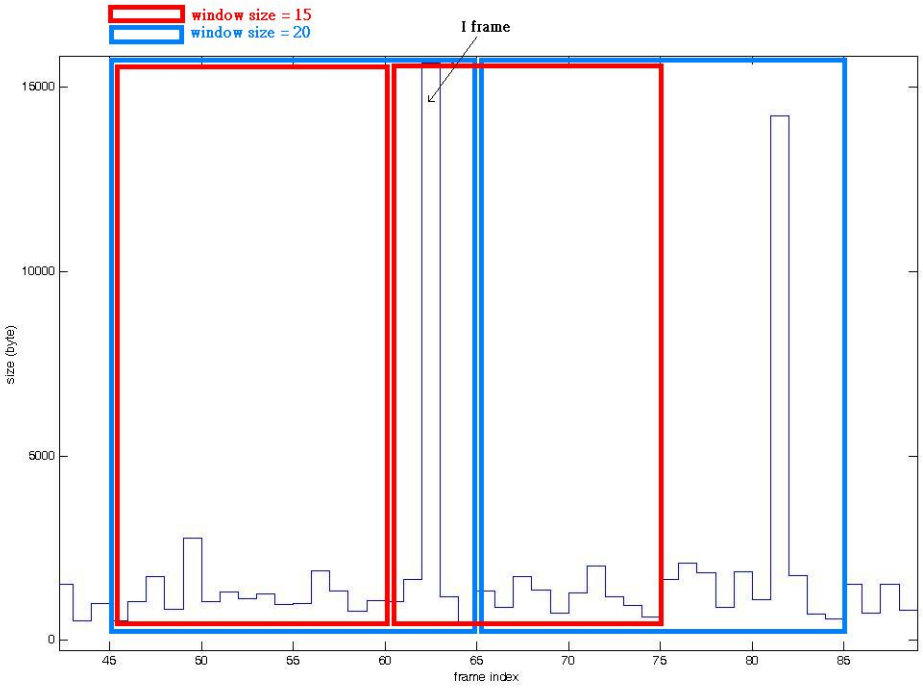


圖 3.1 第二個紅框 window 無法先傳 I frame 的情況

圖 3.1 是一段 MPEG-4 影片中的某 50 個 frame 的大小，如果 W=20(藍框)， 第一個藍框 window 經過 MVBA 運算後，位於框內最後面的那根大資料(I frame)就能預先分擔掉;若 W=15(紅框)，因為 I frame 出現在第二個紅框 window 的前面部份，而且每個 window 是彼此獨立的，所以沒辦法在前一個 window 提前傳送.

還有另一個缺點是，在 window 的第一個 frame 和最後一個 frame 都會形成 critical point. 因為 MVBA 就是要對輸入的 N 個 frame 的 D(t)，找出一條介於 D(t)和 B(t)之間，從 0 到 D(N)的曲線，所以在 0(曲線的起始點，第一個 frame)和 N(曲線的終點，最後一個 frame)都會有個 critical point. 但其實這兩個 critical point 在 online 的情況下是不一定必需的，而多餘的 critical point 會造成 smoothing 效能的下降，從圖 3.2 可以看出來.

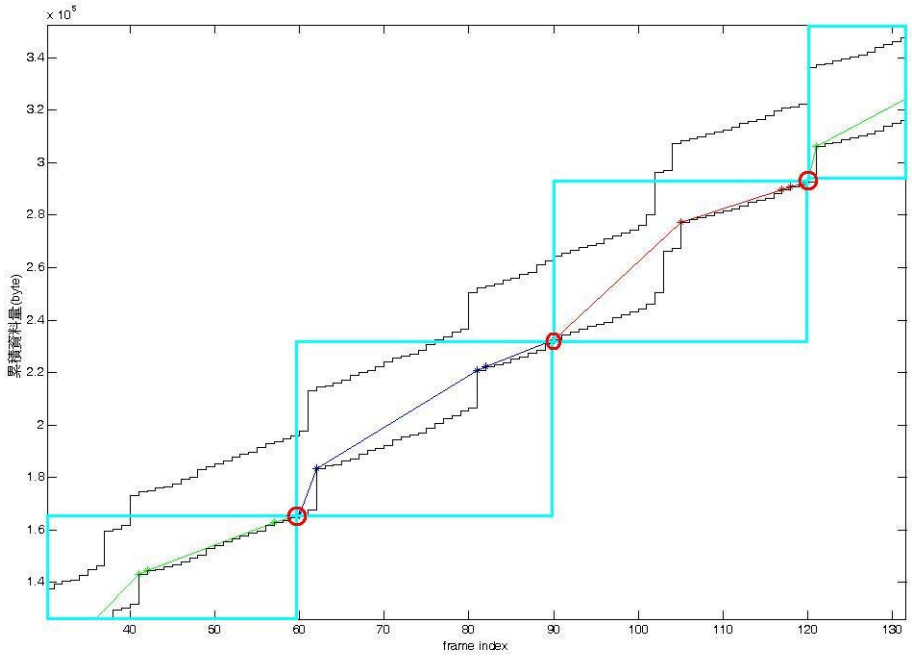


圖 3.2 non-overlapping-window 造成多餘的轉折點(紅圈)

圖 3.2 是個用 hopping-window smoothing 演算法規劃傳輸速率的例子，本例的 window size 為 30 個 frame，一個青色的框框即代表一個 window. 綠，藍，紅三種顏色的細線是規畫出來的排程，之

所以用不同的顏色畫線，是為了區分出是在不同的 window 中規劃出來的.

我們可以發現，window 與 window 之間的 critical point 就算拿掉也不會造成 underflow 或 overflow. 就如圖 3.3 的粗紅線所規畫的線段.

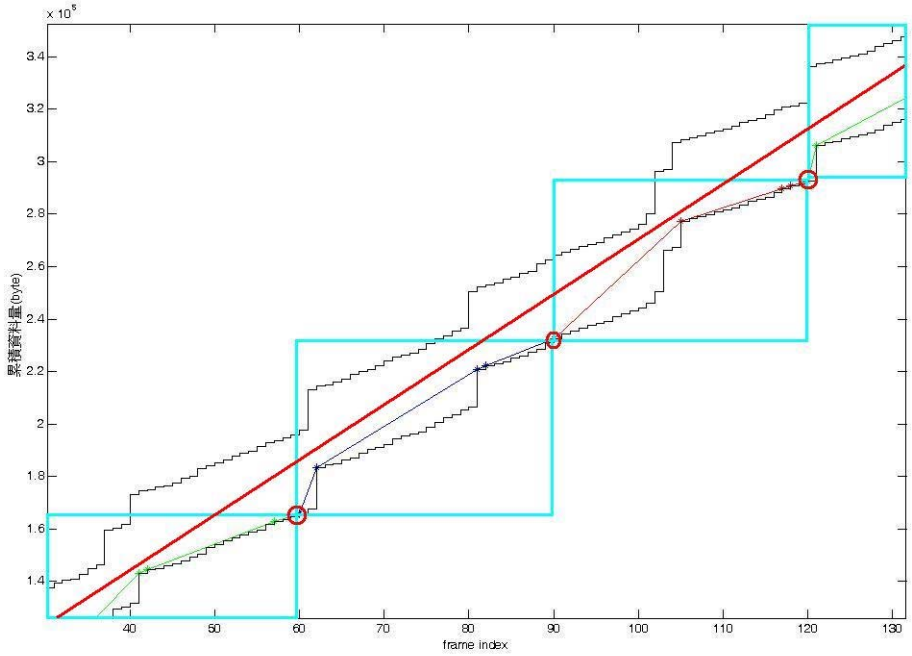


圖 3.3 粗紅線是沒有做 window 切割的 smoothing 規劃結果

從圖 3.3 中可以看出，沒有用 window 切開的規劃(粗紅線)比 hopping window 的方法平順(綠/藍/紅細的細線). 也就是說，本來可以用一個速率滿足要求的情況，因為用 window 隔開而造成了多餘的轉折點，所以必須分成許多段的速率來完成規劃，這麼做的壞處是會拉高整體傳輸速率的峰值 peak rate. 原因如圖 3.4

若 AB 線段是我們原先規劃出的傳輸速率，如今在 AB 間多了一個轉折點 D，我們的規劃的線段就變成 ADB 連線，很明顯的可以看出，AD 的斜率就比 AB 的斜率大，也就是 peak rate 提高了.

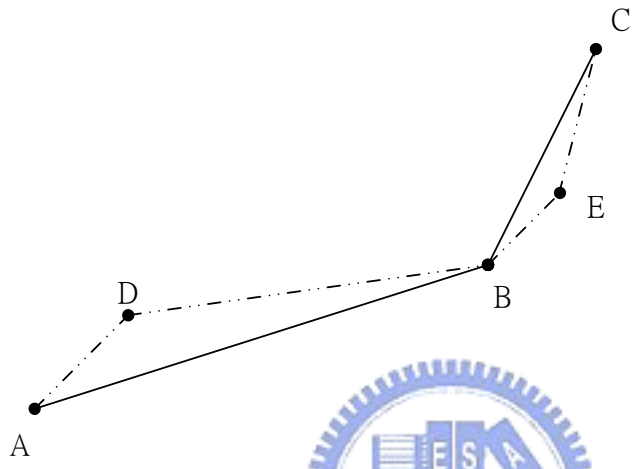


圖 3.4 多餘的轉折點造成 smoothing 效能的下降

再來看有兩段的情況，若 ABC 線段是我們原先規劃的傳輸排程，現在多了個 E 轉折點，而使最後結果變成 ABEC 連線，結果不難發現，EC 的斜率比原先規劃的 BC 斜率大，依然是拉高了 peak rate.

最後做個總結，使用 hopping window 的方法會有二個問題， 第一個問題是每個 window 之間會多出不必要的轉折點，我們把 hopping-window 變成 sliding-window 的方法後就能克服，這在 3.3 小節介紹 sliding-window 時一併說明;第二個問題是,出現在 window 最前端的大 frame 無法預先傳送，這個問題在下面一段文章討論;

第二個問題的原因在於，因為是 online 的傳輸，所以 MVBA 運算

時並不知道這次運算的 window 範圍外，接著會來的 frame 資料量是大或小，但如果傳送的資料是用 MPEG 4 壓縮的話，傳輸資料的大小會有個周期性的規律，特別是資料量很大的 I frame 每隔一個 GOP(group of pictures)會出現一次，所以我們把 window size 設為 GOP 值，然後讓第一個 window 多傳送一個 frame，也就是從第一個 I frame 開始，然後是 BBP… ，最後還包含下一個 I frame.只有第一個 window 其 size 是 GOP+1，第二個以後的 window size 就都是 GOP. 所以第二個之後的 window 會看到 BPP…I，

這樣安排，就能讓第 2 個以後的 window， 在最後一個 frame 才是資料量最大的 I frame.我們設定 W = GOP，並特別設定第一個 window 的 W=GOP+1，結果就能把最大的 I frame 移到 window 的後端.

但如果傳輸的資料不是用 MPEG 4 壓縮的話，就沒有 GOP 可以使用，傳輸資料就不是每隔 GOP 格就會有一個 I frame，那該怎麼辦? 這時可以有另外一種解決的辦法.首先，考慮圖 3.1 所提到的情況，在第 1 個紅框 window 後的第 2 個 frame 剛好是一根資料量大的 I frame，所以在第 2 個紅框 window 做 MVBA 運算的時候，會遇到"在此 window 內，這根 I frame 沒有太多時間可以預先分攤傳送"的問題.

問題的根本原因在於第 1 個紅框 window 在算 MVBA 時，buffer 裡只有 W 個 frame 的資料，還不知道那根大 frame 的存在，所以無法事

先規劃傳送.但實際上，第 1 個 window 剛算完 MVBA，過了 2 個 frame 時間長度後，buffer 裡就已經放了那根資料量大的 frame，如果我們能提前在第 2 個紅框 window 之前多計算一次 MVBA，就能考慮到已經放在 buffer 裡的大 frame 了.也就是說，本來是每隔 W 個 frame 時間長度要算一次 MVBA，現在為了解決"放在 window 前面的大 frame 無法分攤傳送"的問題而要加班，在兩次 MVBA 的計算間隔內再多算一次 MVBA.假設我們是在 W/2 加班一次，那這次看到的 window 就如圖 3.5.

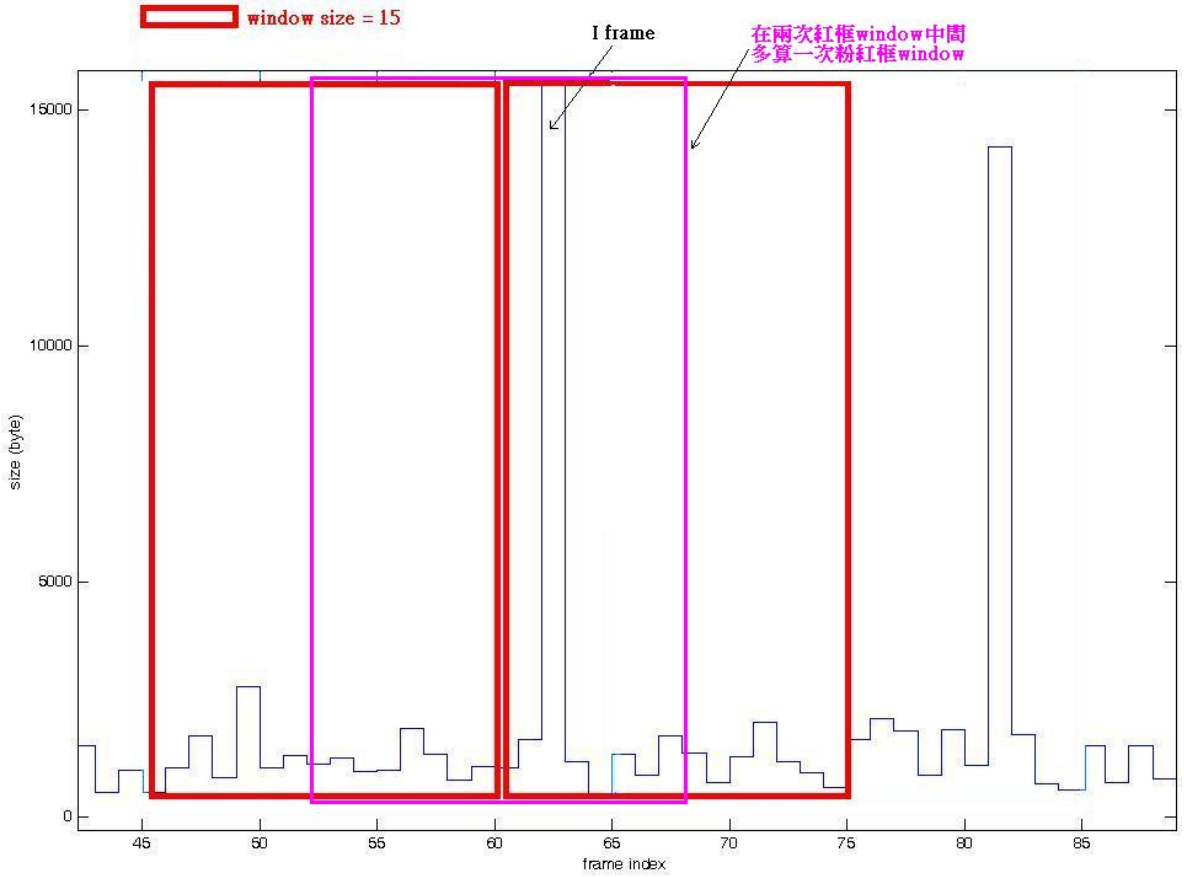


圖 3.5 在兩次 MVBA 的計算間隔中多算一次 MVBA.

在這裡有兩個地方要考慮:

(1)我們該在兩次 MVBA 之間的哪個時間點多算一次?

(2)假設第 m 個 window 的開始時間為 0，且在 W/2 加班一次，則產生的傳輸流程是 smoothing W/2 到(W/2)+ W 之間的 frame，和第 m 個 window 的範圍 0 ~ W，以及第 m+1 個 window 的範圍:W+1 ~ 2W 都有一半重疊，也就是同一個 frame 的時間會有兩種結果，那我們該怎麼決定最終的傳輸排程?

先討論問題(1)，如果我們把加班的時間設置在靠近算第 m 個 window 的時間點，例如，在算完第 m 個 MVBA 之後的下一個 frame 就加班;這樣做的缺點就很類似第 m 個 MVBA 計算所遭遇的困難:對於未來的視野較窄.也就是比起第 m 個 window，這次加班只在 buffer 裡多看到了一個新的 frame;如果把加班的時間點放在靠近第 m+1 個 window,例如,放在第 m+1 個 MVBA 的前一個 frame.則缺點就如第 m+1 次 MVBA 所碰到的:如果資料量大的 frame 是出現在第 m+1 號 window 的第一個 frame，那這次加班可以分攤大 frame 的時間很少，只比第 m+1 次 MVBA 多一個 frame,意即只有一個 frame 可以分擔大 frame 的負擔.做了這樣的比較之後，發現 W/2 是個折衷的選擇.

問題(2)會在接下來的 3.3 和 sliding-window smoothing 演算法一起討論.

3.3 Sliding-window smoothing algorithm

在[3]有提到一種叫 sliding window 的演算法，也可以解決上一節的問題(1).其做法也是定一個 window size 為 W 的 window.接著每隔 alpha 個 frame 的時間長度就重新計算一次,就像是把這個 window 滑動 alpha 個 frame，然後做一次 smoothing 的運算，其中 1<alpha <W，圖 3.6 是 Sliding window 運算的時序圖.

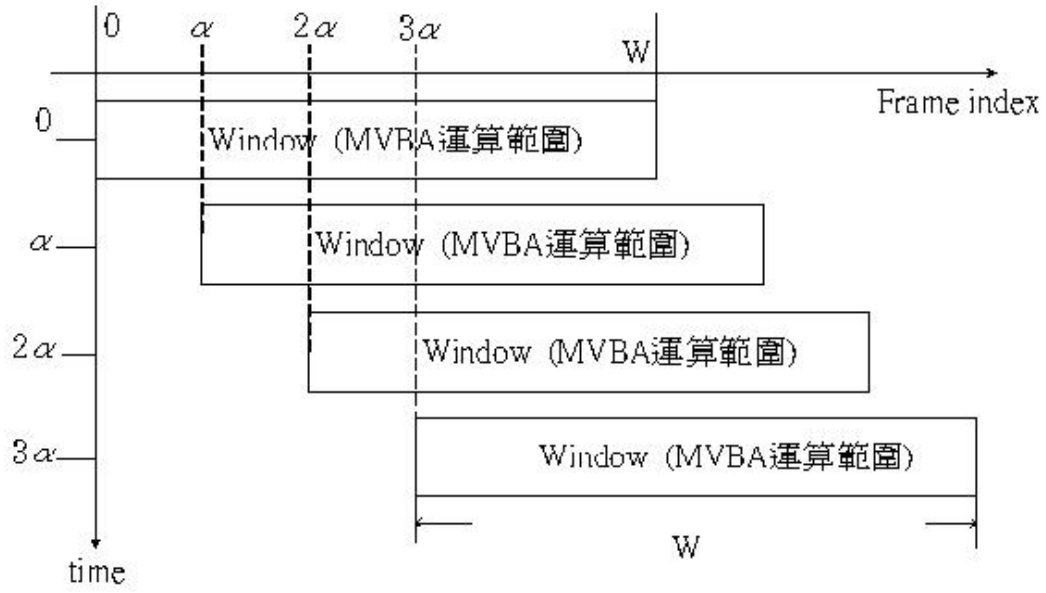


圖 3.6　Sliding window 運算之時序圖

在 alpha=1 的情況下，就相當於 buffer 每進來一個 frame，就重新計算一次 smoothing 的排程.這樣做會用掉大量的運算量.如果 alpha=W，就變成 hopping-window 的方法;若 alpha=W/2，其演算法就剛好是前一小節提到的方法，就是在兩次 MVBA 中間 W/2 的地方再加班一次的做法.然而[2]的模型比較通用 ，所以我們採用這個模型.但[2]只提出 sliding window 的概念.並沒有詳細說明如何將 stored video 的 smoothing 演算法改進成適用於 online，這部份就是我們這

個章節要說明的.

現在來回答 3.2 提到的問題(2):圖 3.6 中，從 alpha 之後的每個 index 都有重複算了 2 次以上的 MVBA，也就是同一個 index 會有兩種以上的排程結果，那我們該怎麼決定最終的傳輸排程? 最初沒有注意到這個問題，把 2 次 MVBA 求出的轉折點都保留下來最後會有如圖 3.7 的情況發生，竟然規劃出負的傳輸速率

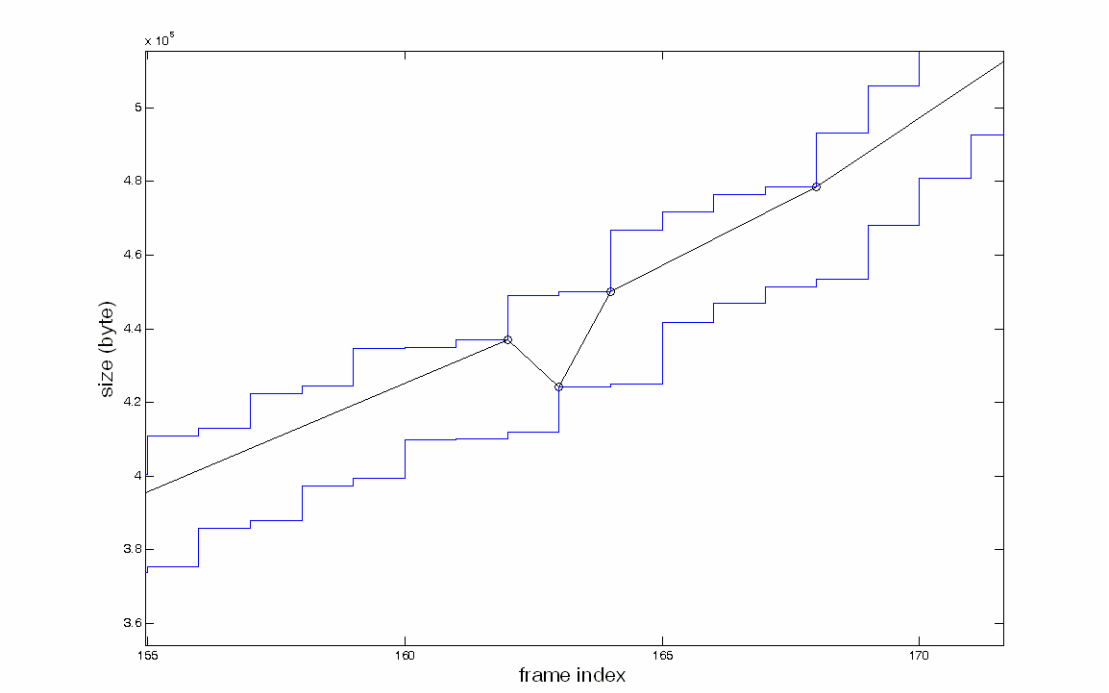


圖 3.7 混合了多次 MVBA 求出的轉折點，造成了負的傳輸速率

以圖 3.7 中的例子來說，因為做兩次 MVBA 有可能算出前一個轉折點在第 162 個 frame index，且是在 upper bound 上，而後一次 MVBA 算出的轉折點在第 163 個 frame index，且是在 lower bound，最後這兩次的結果都留下來，就會產生負的斜率.

合理的做法應該是以最後運算的結果為主，因為最後一次運算有考慮到最新收到的 frame 資訊，所以算出的結果也會最好.如果只留新的，舊的都不要，也會有問題.重點在於 MVBA 會以 window 的最前端的 lower bound 為起始轉折點，但實際上這樣就沒有考慮到前一次 MVBA 所規劃出來的流程.等於是無故在 window 的起點加了一個在 lower bound 上的轉折點，這樣也有可能造成負的傳輸速率.

為了避免這種情形，每次計算 MVBA 時，要從該 window 範圍的前一個 critical point 出發，因為 critical point 本身就是個起始點，從此出發，就不會無故在 window 的起點多加一個在 lower bound 上的轉折點.在程式碼中，我們多加了一個儲存前次 critical point 的變數 ts_last.每經過 alpha 個 frame，就會運算一次 MVBA，而運算結果只決定了前 alpha 個 frame 的傳輸排程(alpha 之後的排程會由之後的 MVBA 做決定).所以 ts_last 這個變數負責記錄 MVBA 的結果中，位於 window 的前 alpha 格內的最後一個 critical point.這樣做能成功消除負斜率的產生.

上一段敘述中，我們決定"從該 window 範圍的前一個 critical point 出發"，這裡忽略了一點:對已經送出的封包做規劃沒有意義.也就是說，在這次 window 之前的時間，已經按照前一次規劃好的排程傳出封包了.要從前一次 critical point 的時間點重新規畫，就算

做出了更平順的規劃,也沒辦法實行,因為那是一段已經過去的時間.為了更清楚地解釋，我們先看一下每個 window 能"有效地"決定的排程時間:如圖 3.8

基本上有效範圍就是從該 window 的起始點後的第 1~第 alpha 個 frame.接著來看一個實際的例子，如圖 3.9

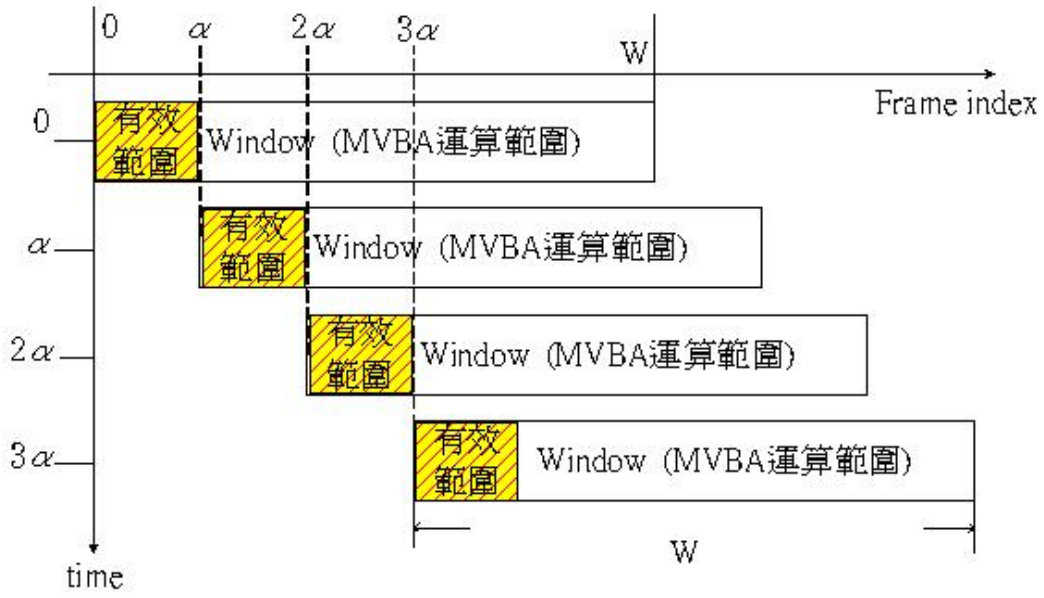


圖 3.8 每個 window 實際決定的排程時間

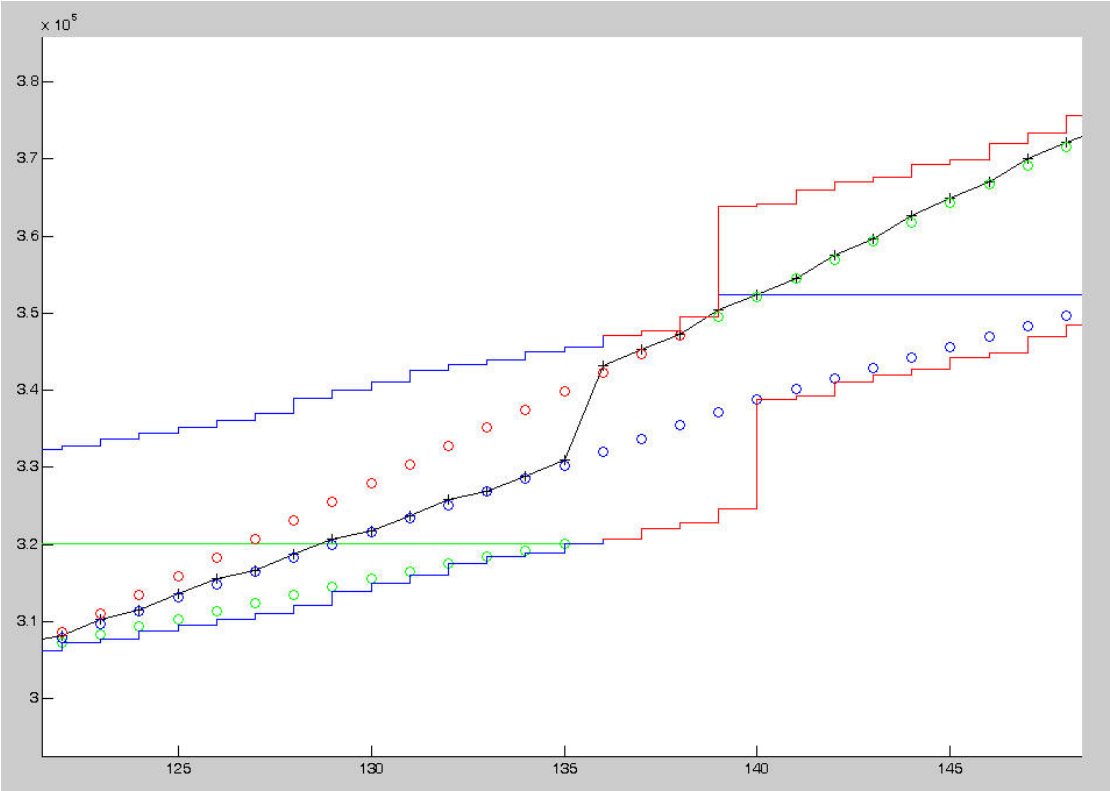


圖 3.9 同一區間有 2 個 window 排程的例子

圖 3.9 中標線有些複雜，先解釋一下. 我們的參數是取 Window size = 30， alpha=15. index120~150 是一個 window. 算出的 MVBA 以藍色的圈圈表示，真正會依照藍色圈圈的規劃傳送的時間是 index120~135; 而 index135~165 是下一個 window 的範圍，算出的 MVBA 以紅色的圈圈表示. 從 index135~150 這段時間會按照紅色圈圈所安排的曲線來傳送. 黑色的實線表示最後決定的傳輸排程.

圖中可看出，黑色實線的前半段和後半段在切換 window 時有落差，而造成一個像斷層般的線段(index135~136). 這條忽然增大的傳輸斜率是我們不樂見的. 這段暴衝的起因是，我們想在下一個 window 的第一個 frame 時間就想補足紅圈比藍圈高出的量，若將這段落差分

期付款慢慢補足，就能減緩暴衝的程度.

那麼要怎麼分期付款這個斷層才平順?這個問題換個問法，也就是怎麼把這個斷層的資料量，用一個平順的傳輸速率傳出去，而且不會有 underflow 或 overflow 的情況?問題的答案仍是 smoothing.當然，我們本來就是在做 smoothing，如果能同時把這個斷層問題解決就很完美了.

想得更仔細一點，這問題的起因是，從 hopping window 演化成 sliding window 時，在 window 重疊的部份，該怎麼銜接的問題.現在的 window 要接著前一個 window 進行到一半的規劃,就一定得把前面 window 已經傳了多少資料這個參數考慮進來;所以,若硬是拘泥於"MVBA 只能從 lower bound 出發"，就註定會面臨斷層的煩惱.

我們放棄從前面一個 critical point 出發的方法，而改從這段 window 的起始點開始，並接續上一個 window 已完成的傳輸量，也就是把起始 critical point 放在半空中.這樣的做法已經更改到 MVBA 的演算法，但也只是修正了起始點，在起始 critical point 之後的運算方法都和原來的 MVBA 相同.其實這個起始點就是式 2.4 及 2.6 裡的 q 值.經過這樣的修改之後，本來圖 3.9 有 burst 的情況即得到改善，如圖 3.10

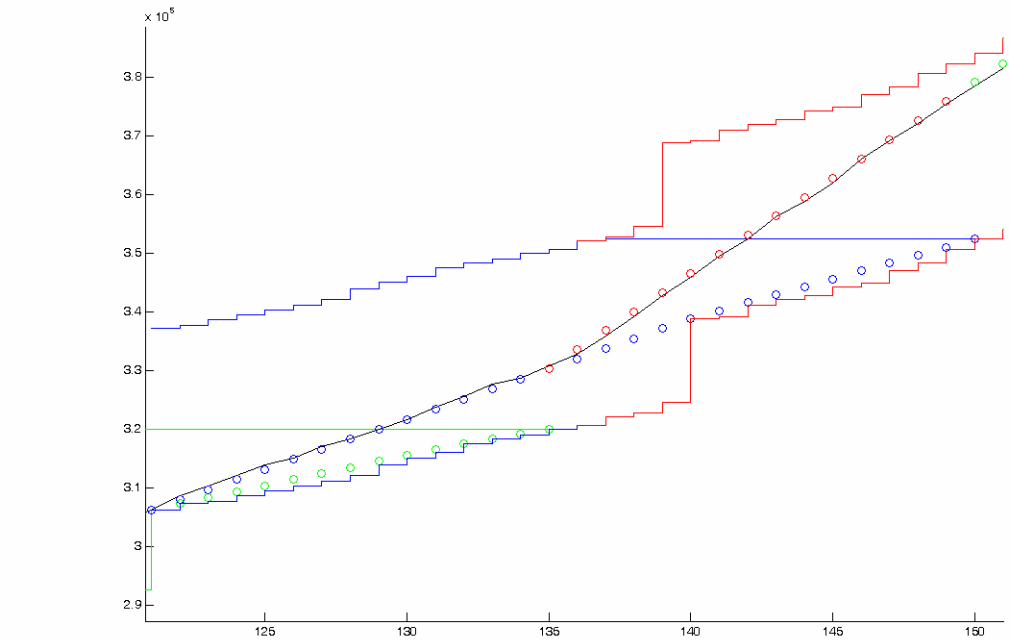


圖 3.10 以前一個 window 運算結果當起始值

3.2 節介紹 hopping-window 的方法時，有談到 window 和 window 之間的連接點會造成多餘的轉折點.在改用 sliding-window 的方法之後，不再是前一個 window 終點接在下一個 window 的起點，也就沒有這些多餘連接點的問題了.當然，這邊說的 sliding-window 是令 alpha<W，否則 alpha=W 的情況下， sliding-window 會變成 hopping-window.

最後我們做一個總結.sliding window 有兩點需要注意的地方，第一點是每個 window 的起始點該怎麼選取;再來就是 window 重疊的地方，該怎麼決定用哪個 window 的規劃的排程當作最後的排程.

若把 MVBA 的起始點設在該 window 的第一格的 lower bound 上，

會造成多餘的 critical point.若不想有多餘的轉折點，則必須要以轉折點為起始點，最適合的選擇是最靠近此 window 前的第一個轉折點.如圖 3.9 中，紅色 window 的 MVBA 運算範圍是在 index135~165.但是取在 index135 之前最近的一個轉折點當做起點,也就是在 index 121 的 Lower bound 上的點.這樣做出來的規劃避掉了在 window 起始點上的多餘轉折點.

在以上的做法中，當時間在 index135 時，紅色 window 開始計算 135~165 的傳輸排程時，index 120~135 已經按照之前藍色 window 規劃的排程傳送了，若要在 index135~160 按照紅色的規劃來傳，會有一個落差產生.這是因為紅色 window 沒有考慮到藍色 window 在交接處已經傳送了多少資料的緣故.

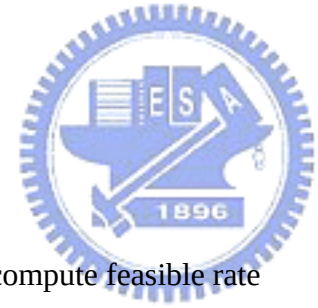
經由以上的討論，window 的起始點就看前一次 window 最後傳送到哪,再由該點繼續規劃此次的 MVBA.在圖 3.9 的例子中,紅色 window 的起點就看藍色 window 在 index 135 時已傳送了多少.

總之，Window 有重疊的地方，讓後算的結果蓋掉前面算的結果.但是後算的 window 的起始點需要看前面 window 算出來的結果來決定.這個結果就是在第二章的 2.4 – 2.7 式中的 q 值.

3.4 MVBA 之修正

另外，我們觀察到，若按照[13]所敘述的 MVBA，則在處理到末端的時候，可能會產生不必要的 critical point. 之前使用在 stored video 的情況下，因為整段影片只做一次 MVBA，只有一次遇到 window 末端的機會，所以這個問題並不嚴重. 然而在 online 的情況下，每幾十個 frame 就是一個 window，這些在 window 末端多出的 critical point 就成為很可觀的累贅. 所以我們對演算法再做了小修正，以適應 online 的應用環境.

原本的演算法 Pseudo-code 如下圖:



```
//set inital values
1   ts = 0, te =1, t_B =1, t_D =1
2   while (ts ≤ N)
3   {
     //advance one index and compute feasible rate
4     te' = te +1
5     if ( we can't find feasible rate, and the buffer will be underflow)
6     { output a segment(ts~t_B); let next ts = t_B, te=ts+1 }
7     else if ( we can't find feasible rate, and the buffer will be overflow or te'==N)
8     { output a segment(ts~t_D); let next ts = t_D, te=ts+1 }
9     else
10     {
11       compare and save the feasible rate C_max, C_min;
12       if C_max, C_min are updated, save the corresponding critical point(t_D, t_B);
13       te = te';
14     }
15  }
```

圖 3.11 MVBA optimal smoothing algorithm pseudo-codes

我們先說明原本的 code 有什麼問題. 首先，演算法會以 ts 時間

當起點，以 ts 到 ts+1 的最大/最小傳輸速率當成目前的 Cmax/Cmin，然後計算 ts 到 ts+2 的最大/最小傳輸速率 Cmax’ / Cmin’，然後取 Cmax 和 Cmax’中較小的當新的 Cmax;取 Cmin 和 Cmin’中較大的當新的 Cmin.總之，Cmin/Cmax 是記錄著可採用的最小/最大傳輸速率.這邊的問題在於，第 2 行的 while 迴圈會執行到 ts=N，（總長度為 N 個 frame，所以 N 是最後一個 frame).因為要計算 Cmin/Cmax 和 Cmin’/Cmax’，就需要 N+1 和 N+2 兩個時間點的資料，而這兩點的資料已經超過影片的最後一個 frame，所以是不存在的.所以我們最多只能讓 ts 跑到 N-2 為止. 我們需將 while 的條件改成 ts ≤ N-2.

再來我們還發現，ts 最大只能定到 N-1，也就是轉折點最多能決定到 N-1 這點.而且 N-1 這點也必定是轉折點.原因解釋如下:演算法一直重複運算 while 迴圈，當進行到最後幾次迴圈時，假設當時 ts ≤ N-3，且 te 前進到 N-2，te’=te+1=N-1，而剛好在 te’這點新算出來的 Cmin’比 Cmin 大，td 變成 te’，也就是 td=N-1.while 做下一個迴圈時，碰到第 7 行 te’==N 這個情況，就會讓 ts=td，並定 td 為轉折點，又此時的 td=N-1，所以 N-1 變成轉折點，而 ts=N-1 已經不在 while（ts ≤ N-2)的條件內，所以迴圈終止，轉折點只定到第 N-1 點，我們要自己在第 N 點加上一個轉折點，連出最後一條線段.

若 N-2 也變成轉折點時，也就是 ts=N-2 時，td 的初始值會給成

ts+1，也就是 N-1，接著馬上碰到 if (te'==N)這個判斷式，演算法會讓 td 也變成一個轉折點，即 ts=N-1;然後 ts > N-2，演算法終止.所以我們必須自動在第 N 點也加上一個轉折點,以完成最後一段 CBR.

在 N-1 必有轉折點的現象當然是不好的，會在每一個 window 的末端產生不必要的轉折點.最不好的情況是，在第 N 個 frame 大小比第 N-1 高很多，(第 N 點是 I frame 的情況).如圖 3.12 中標示"修改前"的藍色實線，在最後會有個突然增加的速率.

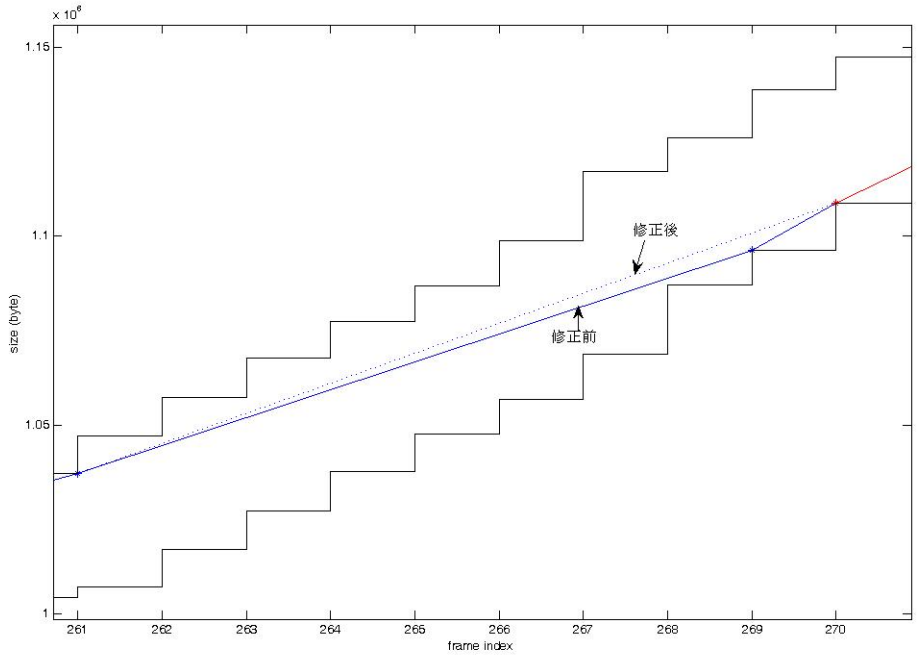


圖 3.12 在 frame index N-1 一定會有轉折點，平順效能降低

我們要修正在 N-1 必有轉折點的現象，code 修改如圖 3.13，我們只是在 te'前進到 N 的情況時，多加了一個判斷(第 9~11 行紅色的部份)，如果 ts 到 N 的斜率是可行的，那就直接以 N 為轉折點.修改

後的模擬結果就如圖 3.12 中標示"修改後"的虛線，確實得到比較平順的排程曲線.

```
//set inital values
1   ts = 0, te =1, t_B =1, t_D =1
2   while (ts ≤ N-2)
3   {
     //advance one index and compute feasible rate
4     te' = te +1
5     if ( we can't find feasible rate, and the buffer will be underflow)
6     { output a segment(ts~t_B);let next ts = t_B,te=ts+1  }
7     else if ( we can't find feasible rate, and the buffer will be overflow or te'==N)
8     {
9       if (te' ==N && the rate of segment(ts~N) is feasible)
10      { output a segment(ts~N);let next ts = N }
11      else
12      { output a segment(ts~t_D);let next ts = t_D,te=ts+1}
13    }
14    else
15    {
16      compare and save the feasible rate C_max,C_min;
17      if C_max,C_min are updated, then save the corresponding critical point t_D,t_B;
18      te = te';
19    }
20  }
```

圖 3.13 修改後的 pseudo-codes

3.5 packet based 之修正

為了在實際的傳輸平台實作 smooth 傳輸，我們必需要將規劃好的傳輸速率轉變成以封包為單位的數值. 因為系統每次傳送都以封包為單位，例如說，第 5 個 frame index 傳 1 個封包(1024byte).

多數情況下，每個封包的大小是一樣的，只有到每個 frame 的尾巴時，因為每個封包要填上所屬的 frame 編號，所以最後一個封包不能同時包含前一個 frame 剩下的資料，又再裝下一個 frame 的資料. 也就是說，每個 frame 切到尾巴會剩下不滿一個封包大小的資料，這剩下的部份就自己當一個封包來傳，這個封包的大小會和一般的封包大小不一樣.所以要以封包為單位修正傳輸排程時，首先我們要先知道每個封包的大小.做法很簡單，把每個 frame 用封包去切割，例如說第 1~4 個 frame 的大小分別是 7428 bytes， 922 bytes， 1960 bytes， 686bytes，以 1024 為單位的封包去切就會變成 11 個封包，大小分別為 6 個 1024bytes，1 個 260bytes，922bytes ，1024bytes，936bytes， 686bytes.記下這一串數字就得到每個封包的大小值.每當我們拿到一個 window 各個 frame 的大小後，就能做如上的運算，求出每個封包的大小，然後針對上下限 B(t)，D(t)做修正.接著依 D(t)，B(t)用 MVBA 算出一條傳輸排程.最後再以封包為傳輸單位，找一條最接近 MVBA 運算結果的曲線.

上述以封包為基底的演算法是在[15]提出的，作者對傳輸的上下限 B(t)和 D(t)做了以封包為單位的調整，而我們認為這個步驟其實並不必要.多這個步驟的缺點是，做上限的修改需要消耗較多的運算資源(比起我們提出的方法). 而 online 傳輸需要的是即時性，為了

降低伺服器運算造成的延遲，我們想用別的方法代替這個步驟.

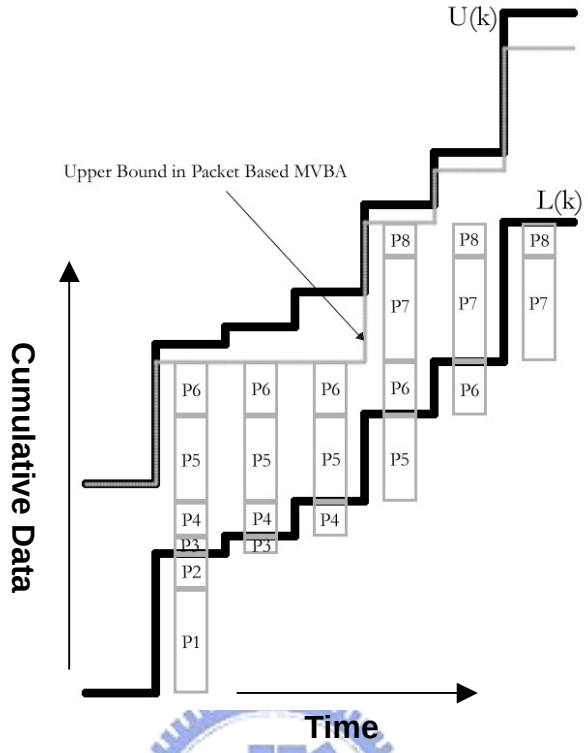


圖 3.14 將上下限以封包為單位做修正

為什麼說以封包為單位修改上限消耗運算資源? 我們先來看看他是怎麼做的: 如圖 3.14，我們要將黑色的上限修正成灰色的，基本上就是每個單位時間跑一個 while 迴圈，不斷比較累加的封包量有沒有超過原本的上限值 B(t). 把封包累加的量記成 B'(t)，累加到很接近但小於 B(t)，若 B'(t)再加上一個封包，就會超過 B(t)，例如圖中前 3 個時間 B'(t)只能累加到 P6，若加到 P7 就會超過 B(t). 這個 while 迴圈看有幾個 index 就要跑幾次，所以頗花時間，而且也還要多一份記憶體來存 B'(t). 以封包為單位對下限 U(t)做調整的話，不會造成改變，因為 D(t)本身就是每經過一個單位時間就增加

一個 frame 的資料量,而一個 frame 的大小剛好會被切成整數個封包.所以 D(t)每過一個單位時間就會增加整數個封包，再經過以封包為基底的修正也不會造成改變.所以只有上限 B(t)做調整時會改變，然而，B'(t)和 B(t)的差異最多也只到一個封包的大小.在 buffer 有 1Mbytes 以上的情況，一個封包 1024bytes 的變動並不會對 MVBA 的結果產生多大的影響.

在[15]中，會需要對上限做封包為單位的修正，是為了避免如圖 3.15 發生的 overflow 情形.

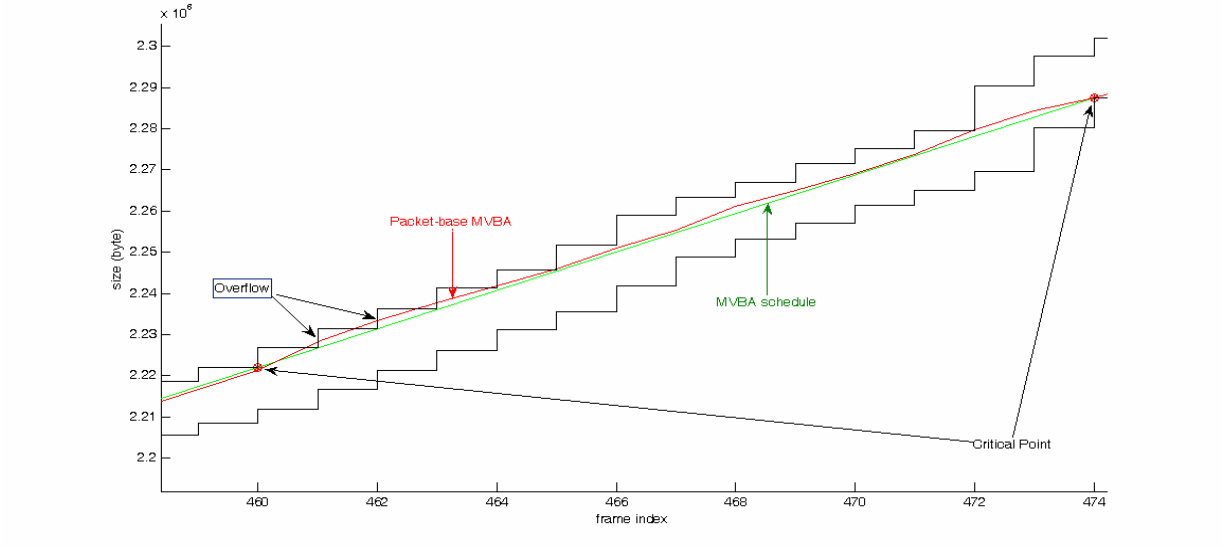


圖 3.15 少了上下限的 packet-base 修正而產生的 Overflow

圖中綠線表示原始 MVBA 做出的傳輸排程，而紅線是以封包為單位所找出最接近 MVBA 的排程.在每個單位時間，最接近原始 MVBA 的選擇有兩種，不是在線段上就是線段下.在[15]的設計中，會跟據下

一個轉折點放在上限或下限分別決定在線段下或是線段上;圖中左邊上限和右邊下限的兩個紅色圓點表示轉折點,在左邊的轉折點是放在上限,所以該點左方的紅線(Packet base MVBA)會在綠線(原始 MVBA)的下方;同理,因為圖右邊的轉折點是擺在下限,所以此轉折點的左邊紅線會在綠線之上.在圖中我們很發現從左邊轉折點出發後不久就有 overflow 的情況產生(index461~463 之間),因為此線段的出發點是放在上限,綠線在短時間內離上限的距離還是很近,此時紅線又選擇要傳送比原始 MVBA 排程還多的量,就有可能 overflow. 先做上限的調整就能防止這種情況,經過上限調整之後的 B'(t)一定比 B(t)小,而跟據 B'(t)算出的 MVBA 排程就一定在 B'(t)以下,就算在最接近 B'(t)的地方,選擇 MVBA 線段上方的封包,頂多也只到 B'(t),而 B'(t)又在 B(t)之下,所以不會有 overflow 的問題.

其實要避免 overflow,不需要先修改上限,只要在做完原始的 MVBA 後,決定最後排程是要在 MVBA 的上或下時,除了參考下一個轉折點是在上限或下限之外,再加一行"if(會 overflow),then 就選下面"的判斷式,就能避免發生 overflow 或 underflow 的選擇.

而用這方法做出的結果,和[15]的做法只有在如圖 3.16 的情況下才會有差異.

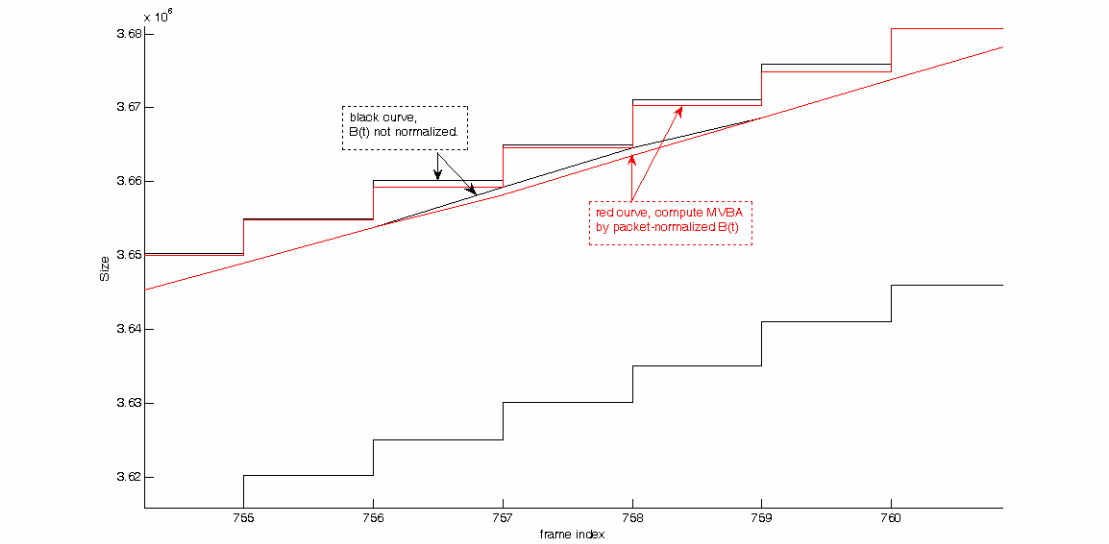


圖 3.16 兩種方法產生的排程差異

圖中紅線是[15]的方法，注意紅色的上限是以封包為單位修正後得到的;黑線則是我們的方法，上限沒有經過調整，所以和原來的一樣.兩種方法規劃出的排程，即紅線和黑線，在貼著上限跑的時候會有機會不一樣，因為[15]的方法改變了上限(會比較低一點)，所以得到的 MVBA 也可能比較低，而造成選取的封包不同，產生的差異就像圖中那樣.

3.6 總結

為了在即時傳輸的情況下能作 smoothing，我們提出了 non-overlapping window smoothing 的概念,把時間軸上的 W 個 Frame 當成一個 window,以 window 為單位做 MVBA;在 stored video 的情況,整段影片只有遇一個開頭和結尾，現在切成很多小 window 之後，每

一段 window 都有一個起始點和結尾，MVBA 的起始點和結尾的處理就需要特別注意.

Non-overlapping window 此一作法會有兩個缺點. 第一個缺點是，位在 window 前端的大 frame 經過 smoothing 後仍然會有暴衝. 另一個缺點是，在 window 的第一個 frame 和最後一個 frame 都會形成轉折點，這些多餘的轉折點會降低效能. 所以我們提出 sliding window smoothing 的概念.

Sliding window smoothing 後，發現原先的 MVBA 在末端處理上有問題，我們作了使其更為平順的修改.

為了能用在以封包為傳輸單位的網路，smoothing 演算法得到的排程也必須修改成以封包為單位. 然而我們採用和[15]不一樣的做法，可以節省運算量，且效果一樣.

第四章 演算法效能之模擬與比較

第二章，第三章介紹過 MVBA 和 online smoothing 之後，這一章我們要來看看，前面講的 smoothing 演算法能帶來多少好處.我們拿幾個電影的 video trace 來當作 smoothing 的對象，比較 smoothing 前後，還有使用的參數不同，得到的傳輸速率標準差(standard deviation)有什麼變化.另外一個觀察的重點在傳輸速率的峰值(peak rate).

標準差和峰值這兩個統計值是我們用來評斷一個平順演算法好壞的指標.Smoothing 的目標在降低傳輸速率的變動量，也就是說，最好能讓所有的時間都用平均速率來傳輸，所以標準差愈小愈好;同理，愈能把峰值降下來，表示 smoothing 的效能愈好，這表示這次傳輸需要的預留的最大頻寬愈小.

在演算法內可以改變的參數主要有:接收端的緩衝區大小，window size，W.還有 packet base 的封包大小,以及 sliding window 的參數 alpha.而在[13]已有討論過緩衝區大小對平順演算法效能的影響.在[15]也探討了封包大小會對效能有何改變.

我們可以預期，online smoothing 會如同 stored video 的 smoothing 一般，效能會隨著緩衝區加大而變好，這應該是沒有問題

的.然而，我們關心的問題在於，online smoothing 的效能有多接近 stored video.還有就是，window size 以及 alpha 的大小會如何影響平順演算法的表現.

4.1 stored video 和 online 之比較

我們用的測試的影像檔是 cindy 的 MTV，其統計資料如下:平均 frame 大小 5067byte.傳送 peak rate 是 22922byte，標準差是 2864.未經 smoothing 處理的 video trace 如圖 4.1，看起來變動很大.

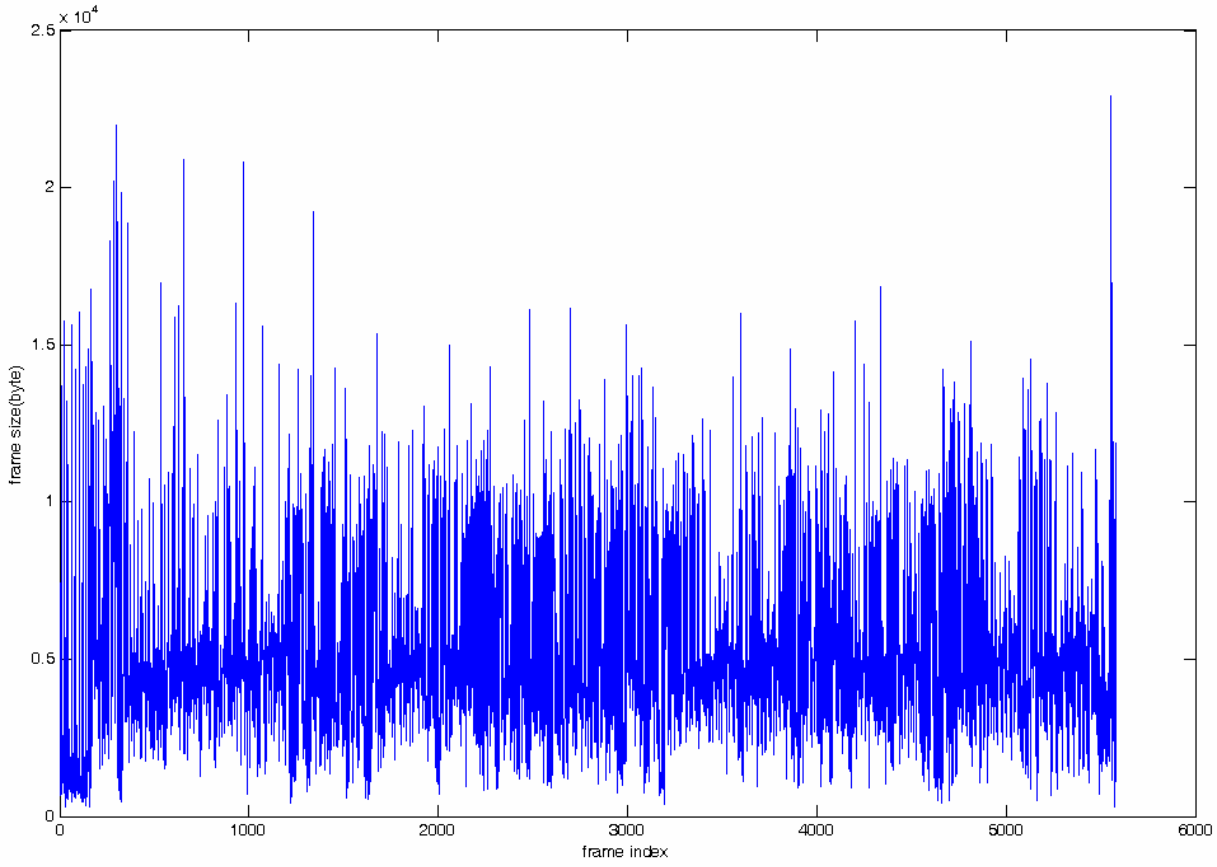


圖 4.1 cindy MTV 的 video trace

接下來我們看看經過 stored video smoothing 和 online smoothing 之後的結果.雖說在 MATLAB 模擬時用的資料的都是存好的 video trace，並沒有 online 的資料，不過我們假想這堆資料是一個一個收到的 frame，每收到 alpha 個 frame 就做一次 MVBA，這樣就能模擬即時傳輸的資料.stored video smoothing 和 online smoothing 的比較如圖 4.2.

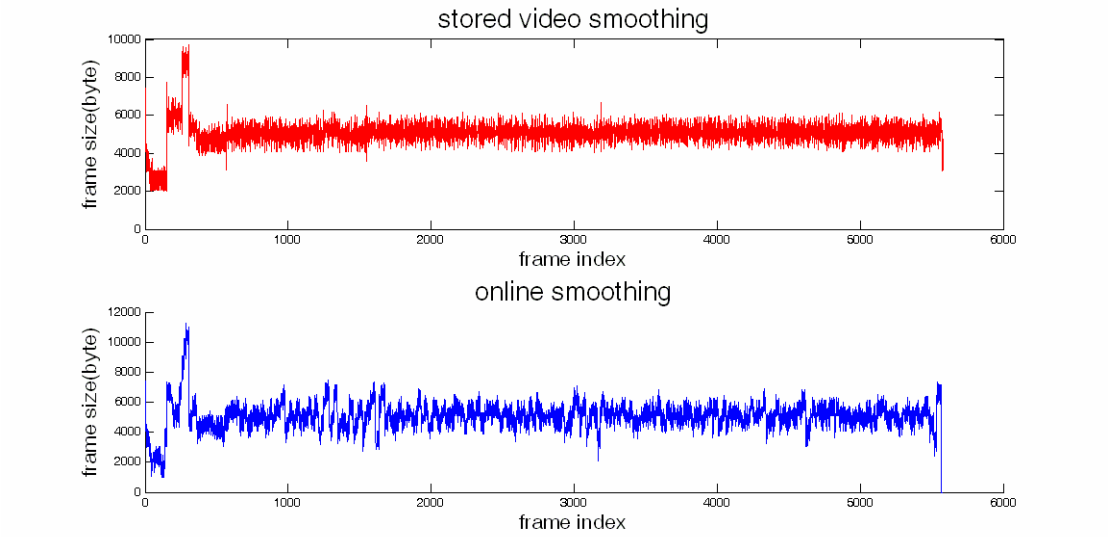


圖 4.2 stored video smoothing 和 online smoothing 的比較

模擬中用的緩衝區大小有 100k,是平均 frame size 的 20 倍左右，算是滿大的，所以除了前面開始時傳輸量太大，無法有效地分攤掉以外，後面傳輸速率看起來多維持在平均速率，也就是 5k/frame 附近.比起圖 4.1 的情況已有明顯地改善.stored video 的標準差縮小到 643; peak rate 也降低到 10585bytes. Online smoothing 效能略遜

於 stored video，標準差是 971，peak rate 是.11264bytes

從時間-傳輸量來觀察，已能大略的看出傳輸量起伏的程度，但不容易比較出 online 接近 stored video 的程度;從理論上來分析，當 online 的 window size 趨近於無限大的時候，就是 stored video 的情況.也就是 window size 大到能存放所有要傳送的 video trace.這就等於像是把即時影像存成檔案然後做完規劃再傳送出去;簡言之，window size 就是平順演算法能嘗試去攤平傳輸速率的視野範圍，若能看得愈遠，就能愈早做規劃，整體效能當然就能提升.然而缺點就是接收端要等更久的延遲才能看到影片.所以加大 window size 雖能增加效能表現,但用戶端延遲也會跟著增加.圖 4.3 和圖 4.4 是 window size 對 online smoothing 效能的影響.

從圖中我們可以看出平順演算法的效能確實是如預期般，隨著 window size 的增加而變好.而且當 window size 增加到 50 之後，不管是 peak rate 或是 standard deviation 就已經慢慢逼近一個極值.我們比較一下 window size 等於整段影片長度的效能(也就是 window size 趨近無限大的情況)，發現 peak rate 是 11318，而 standard deviation 是 918，都很接近極值.若取 Window size=50，則對每秒傳 25 張 frame 的系統而言，server 需要花 2 秒鐘的時間來收集一個 window 的資料，也就是用戶端會有 2 秒的延遲;若是設 W=25，延遲就

只有 1 秒，但效能也很好.我們可以依據應用程式的需要來選擇 Window size.

另外我們發現圖 4.4 的 peak rate 在 window size 等於 40 的附近有不規則的跳動，那是因為 packet base 所造成的影響.我們設定的封包大小是 1024byte，所以 peak rate 的跳動在 1k byte 之內都算正常.

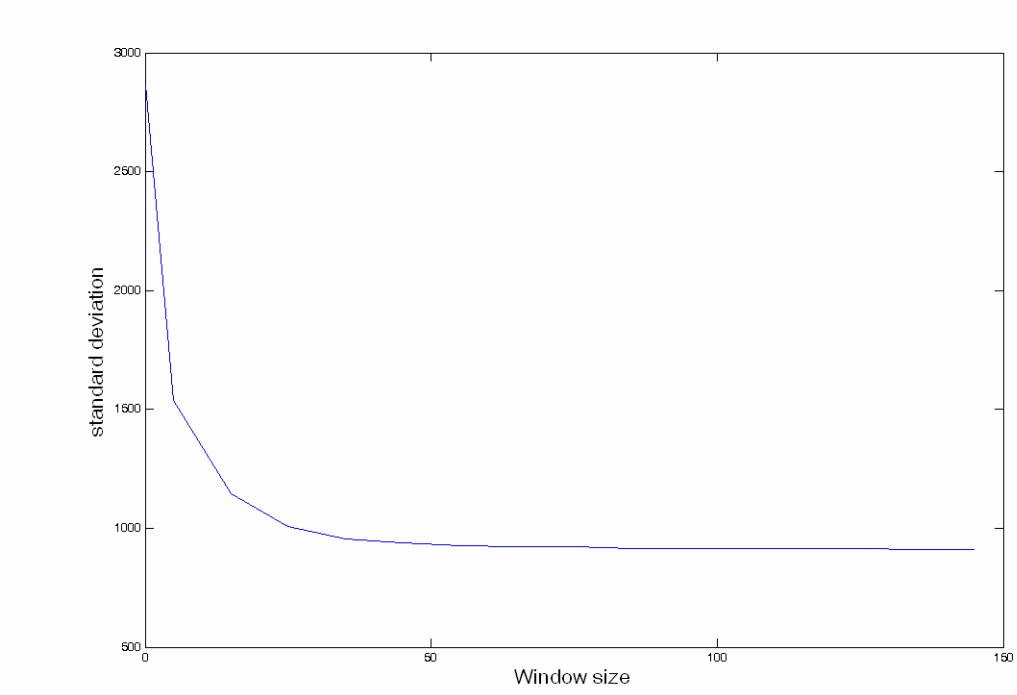


圖 4.3 window size 對 standard deviation 的影響.

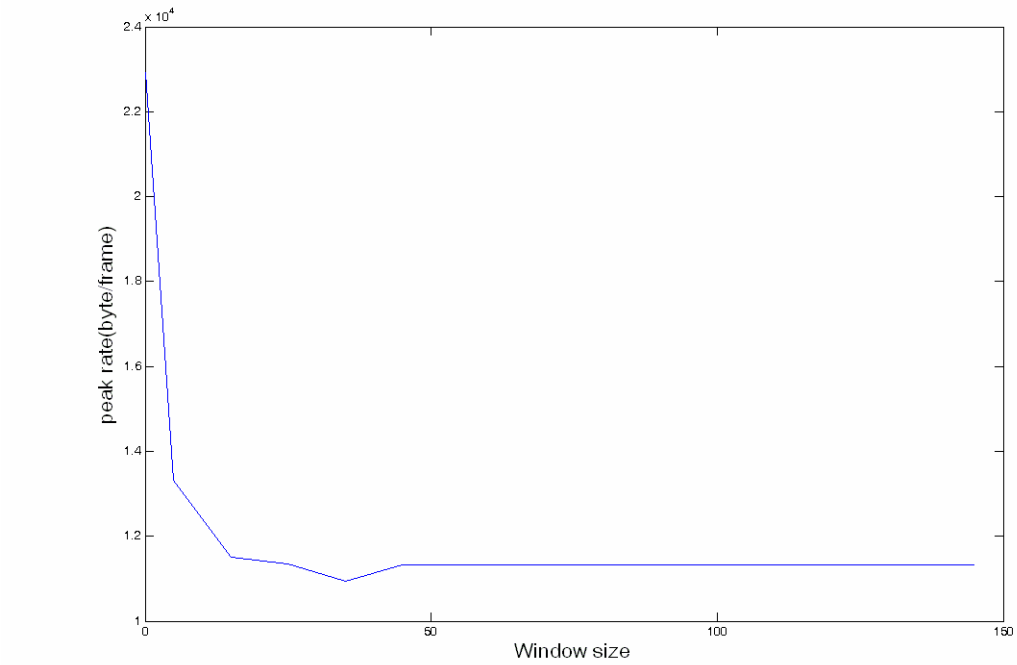


圖 4.4 window size 對 peak rate 的影響

4.2 sliding window 平順演算法參數對效能之影響

在這小節我們討論一下 alpha 對演算法效能的影響，先看一下模擬結果:圖 4.5 和圖 4.6 分別表示 standard deviation 及 peak rate 隨 alpha 變化的情況.

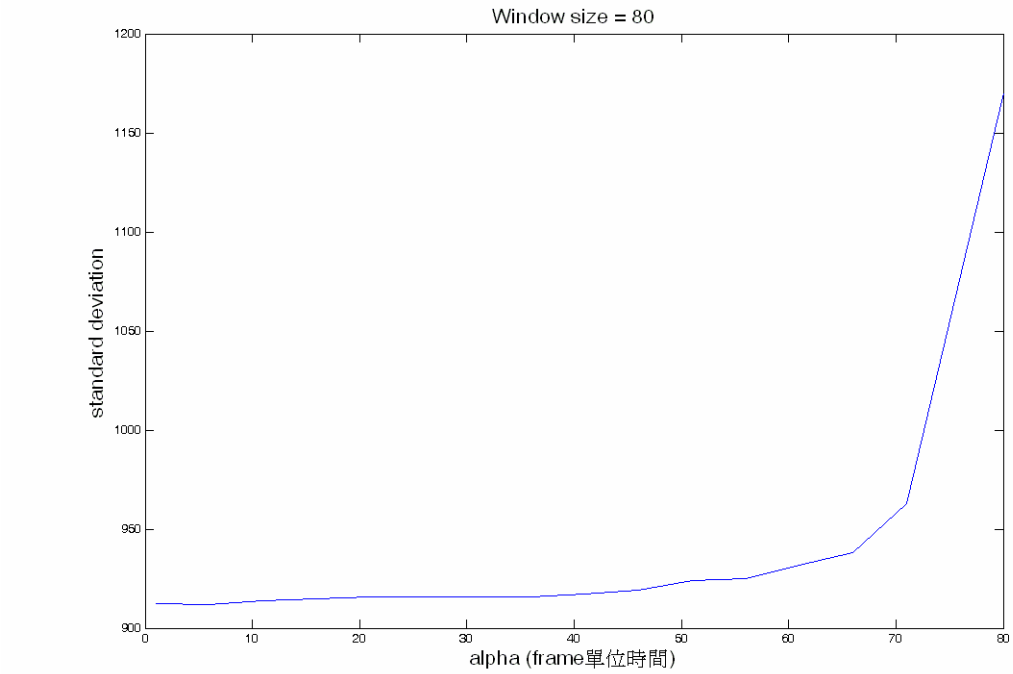


圖 4.5 alpha 對 standard deviation 的影響

這是用 window size 為 80 跑的摸擬，從圖中會發現 alpha 愈小，表現愈好，因為每隔愈小的時間就去看新來的 frame，做 MVBA 運算

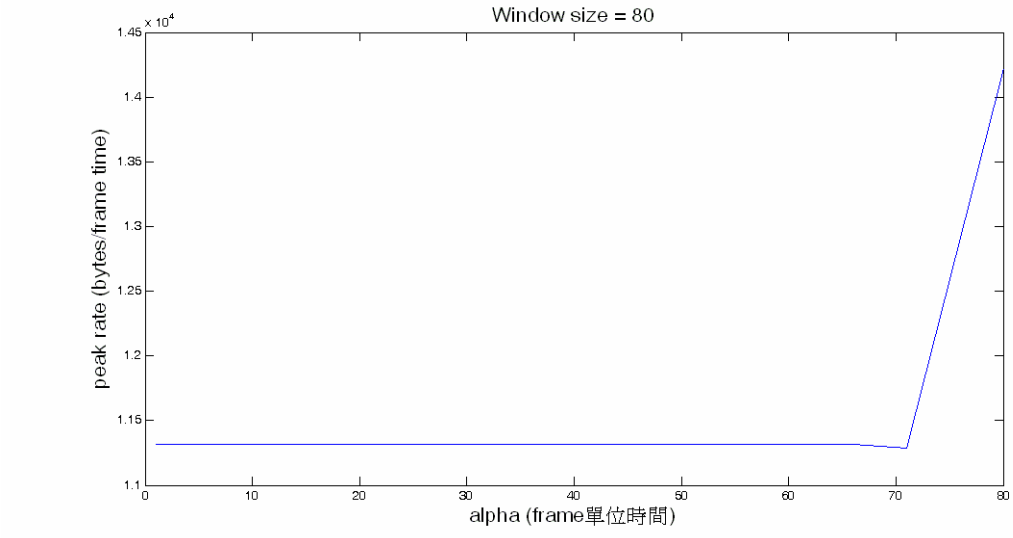


圖 4.6 對 peak rate 的影響

的頻率也愈高，alpha 在 50 以下，其效能已漸漸平穩，所以我們取

alpha=W/2 已足夠.

在此例中，Window size=80 已超過 GOP 數倍，運算的範圍包含了數個 I frame，所以不會有 window size 太小而沒包含到 I frame 的情況(如圖 3.1 紅框).在這種 case 下，alpha 的影響不大.我們比較一下在 Window size=15 的情形，如圖 4.7 和 4.8

當 window size 縮小到 15 時，從圖 4.7 的 standard deviation 可以發現,比較起 window size 是 80 的時候,演算法的效能隨著 alpha 降低的情況更顯著了.所以在 window size 小的情況下，較需要留意 alpha 值的選取.

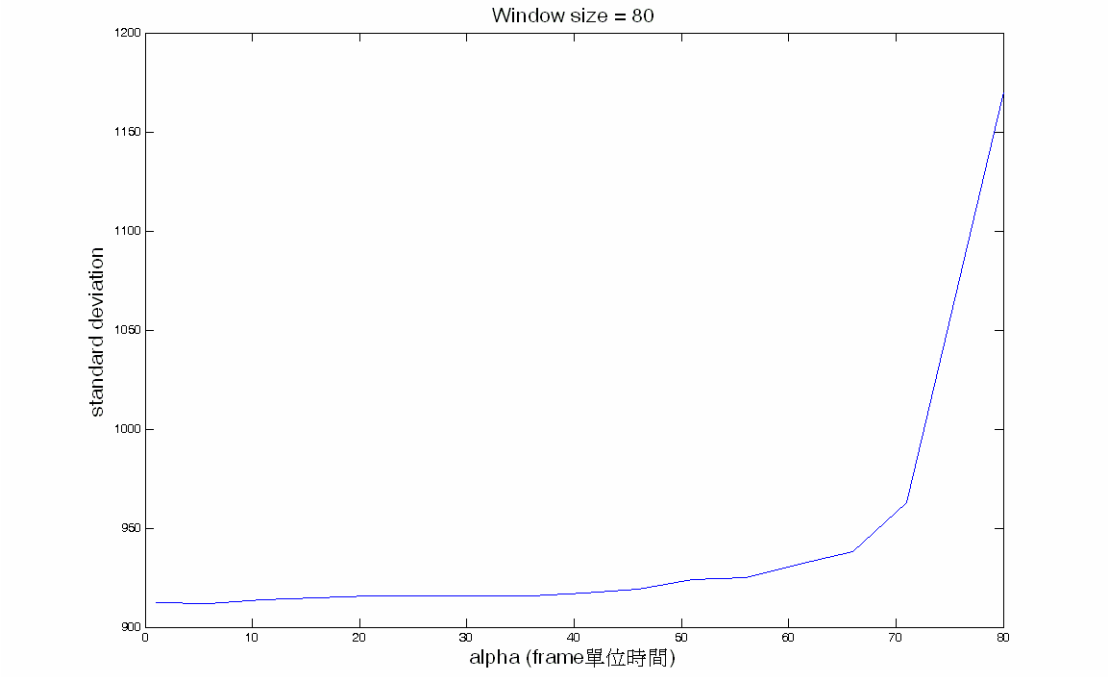


圖 4.7　W=15 時 alpha 對 standard deviation 的影響

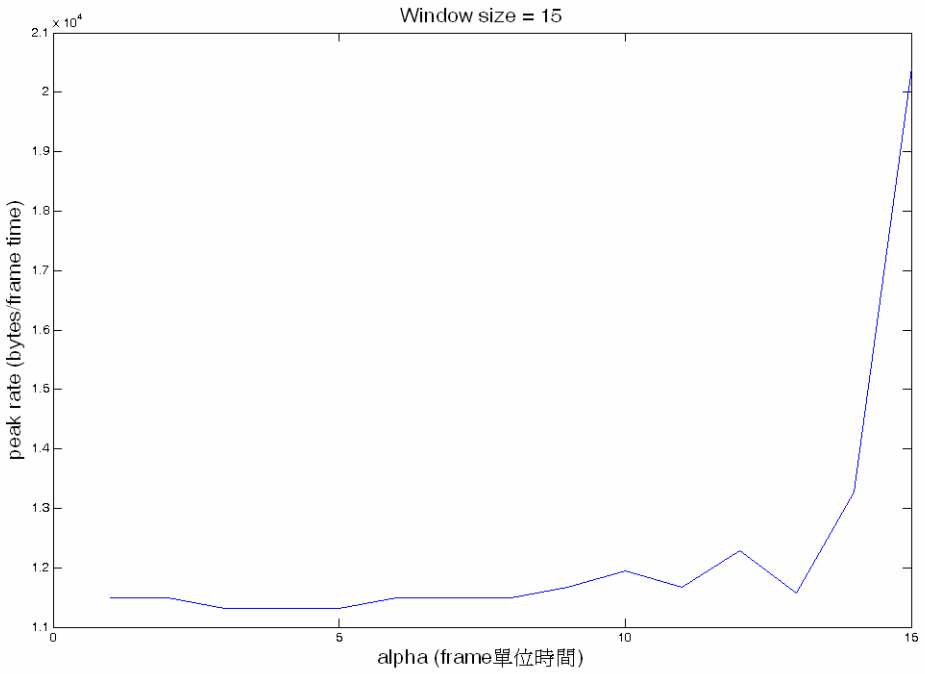


圖 4.8 W=15 時 alpha 對 peak rate 的影響

4.3 總結

在[13]中主要是分析 client 端緩衝區大小和平順演算法效能之間的關係，[15]提出的 packet based MVBA smoothing algorithm，除了緩衝區大小，另外還討論封包大小對效能的影響.

Online smoothing 的效能特性和 stored video 一樣，會隨著 client 端 buffer 增大而變好，而 packet size 則是愈大愈差，因為經過 packet mapping 之後的排程和原本排程的差別之變動就愈大. 從 stored video 延伸到 online smoothing algorithm 之後，sliding

window 的做法有 window size 和 alpha 這兩個參數可以調整.我們發現只要 window size 比 GOP 大，演算法效能就可以有明顯的進步.在 window size 比 GOP 小的情況，可以靠著額外付出的運算(每 alpha 個 frame 時間多算一次)來補足平順效能.然而，在 window size 是 GOP 兩倍以上的情形，alpha 仍然可以增進效能，但是增進的程度不如在 window size 小的那種情況.

第五章 平順演算法之實作與量測

5.1 傳輸平台系統簡介

我們想要將此演算法加到一個實際的串流系統上，在這小節先簡介此系統.

我們的即時影音資訊收集傳送伺服器(以下簡稱 Server)可以將由各種介面收集到的影音資訊及數據資料，透過各種類型的網路(UDP/TCPIP， Wired/Wireless)，即時地傳輸影音及數據資料到需要使用這些資訊的客戶端(client).

圖 5.1 是簡單的系統示意圖:

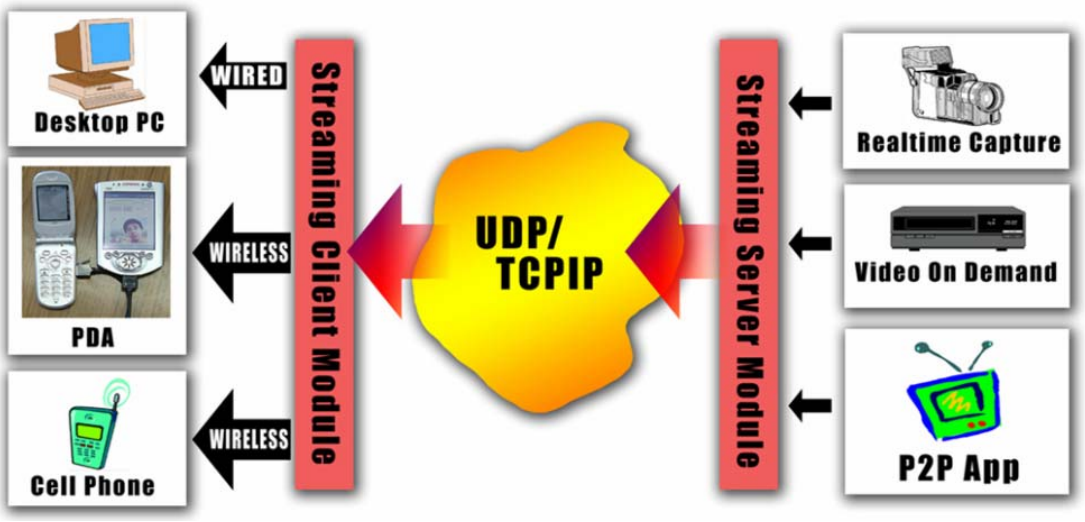


圖 5.1 串流傳輸系統

圖中的右半部，表示 Server 可以即時擷取到攝影機拍攝到的的

影像(Real-time Capture)，然後透過有線網路即時傳送到遠端的電腦，裝在電腦上的客戶端程式能馬上看到攝影機抓到的影像.客戶端程式有很高的可攜性，不管是 PDA 或是手機都能安裝客戶端程式(目前能成功地執行在 WinCe 作業系統上)，也就是能拿著手機或 PDA 邊走邊看從 Server 立即傳來的攝影機影像.

Server 抓取到的影音及數據資料會儲存在自身的儲存裝置中，如果有需要使用的話，可以隨時從 Client 取出來觀看.就像發生竊盜案後，可以調閱案發當時的監視錄影帶.這就是 Video on Demand 應用的一種.

簡單來說，我們的 server 就是一個資料蒐集中心，不管是影像，聲音，或是經由儀器量得的訊號，只要能存在電腦裡，我們就能即時送出，同時也能儲存下來，以備將來使用.

系統功能介紹

這套系統的功能有:1.影像即時傳輸 2.簡單的影像處理(手勢辨視，定位系統)

目前在交大多媒體實驗室已實際架設出這套影像即時傳輸系統，包含了幾個重要的軟硬體，程式內部的工作流程如圖 5.2:

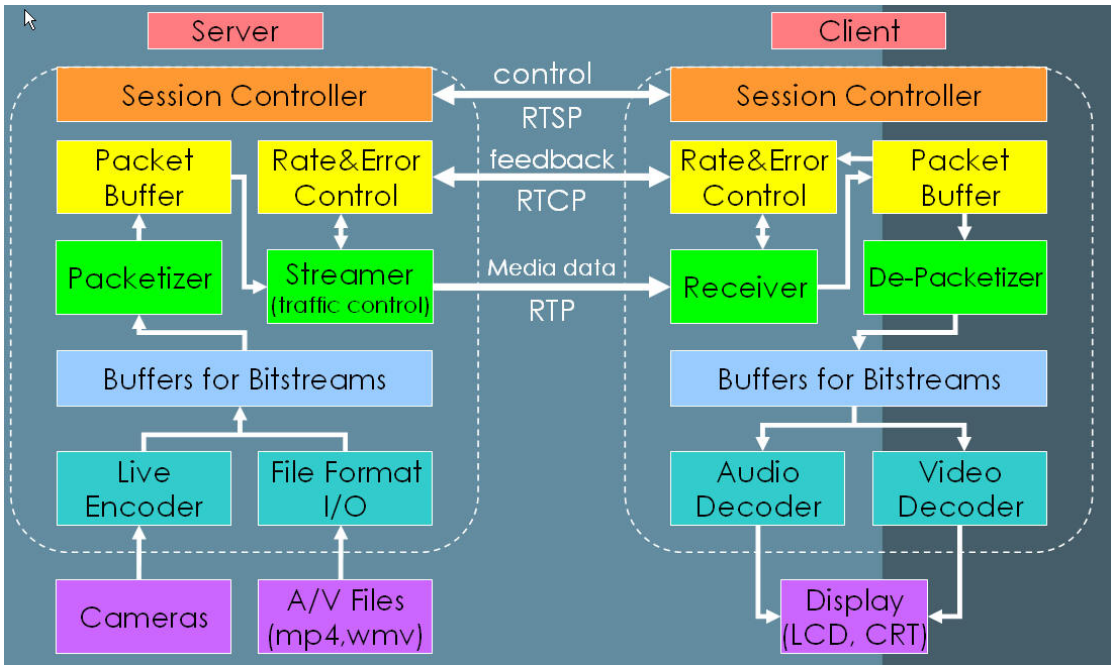


圖 5.2 程式元件及資料流

重要的硬體有左下角的 Camera， 以及左半部的 Server 程式和右半部的 Client 程式.

(1)影像擷取裝置:(Camera)

有八台無線攝影機設置在實驗室各角落.攝影機可以每秒傳送 30 張影像，每張影像解析度為 320 X 240 pixel，攝影機抓到的影像用 JPEG 壓縮後傳到 Server，還可以根據不同需要調整每秒傳 10 張/15 張/20 張/25 張影像,以節省頻寬.另外有數個 USB 介面的視訊 Camera.

(2)即時影像收集傳送伺服器:(Server)

任何有安裝 Server 程式的 Window 電腦都能拿來當作影像蒐集中心， Server 可以任意選取並蒐集那八台無線攝影機經由無線網路傳來的畫面.另外，不只是無線網路攝影機，接在 USB 介面上的小型視

訊 Camera 傳來的畫面也一樣可以抓進我們的 Server 程式裡. Server 擷取到畫面後會自動將連續的畫面壓縮成 MPEG 4 的影片格式，以減少傳送時使用的頻寬.

為了進一步減少頻寬的使用量，我們的 Server 還加入了平順演算法(smoothing)，可以事先規劃封包傳送的流程，把大流量時段的資料分散到其它時間傳送，所以整體的頻寬使用情況會更平順，也更好管理.

Server 和 Client 間的傳輸是採用 RTP/RTCP/RTSP 等專為多媒體通訊設計的通訊協定，這種方式的影音傳輸 我們稱之為串流(Streaming)傳輸，目的是為了能達到即時影音傳輸的要求.

另外 Server 還有 Feedback 的機制，可以根據 Client 的影像處理結果，自動開啟錄影功能. 這套 feedback 機制可以擴充成自動警報系統之類的系統.

在 Server 端 所有的執行緒可以分成三大類

1. 壓縮轉碼單元: Transcoder

負責向各種攝影機即時取像 即時轉碼 即時壓縮成 MPEG-4 格式. 然後把壓縮完的資料放在規劃單元可存取之 buffer 上. 整個 Server 可以轉碼壓縮幾台相機的影像，端看系統的 CPU 運算能力， 整個 Server 只有一個 Transcoder 做轉碼的工作.

2.傳輸單元: PacketScheduler

負責將要串流給"所有 clients"的 RTP 封包，送到網路卡，再由網路卡傳送到網路.因此在整個系統中，只有一個此單元.

3.規劃單元:Streamer， RTSPProcessor

Server 每接收一個新的 Client， 就會準備一組規劃單元， 專門來處理及規劃要傳給這個 client 所需要的封包， 包括了 RTSP 及 RTP 的部份. Streamer 負責將每張影像從硬碟或壓縮轉碼單元所準備之 buffer 抓過來 做 RTP 封包切割的動作.然後將這些 RTP 封包放在傳輸單元可以存取之 buffer 上. RTSPProcessor 負責接收及回應 client 的 RTSP 訊息及執行 RTSP State 的變換.

在這三類執行緒單元中 很明顯地 最花 CPU 時間的是壓縮轉碼單元，因為在這單元中 MPEG-4 的壓縮運算量是相當大的. 尤其是其 Motion Search 的部份佔了將近壓縮運算的 80%，因此，壓縮轉碼單元其實就是此系統最大的瓶頸.

換句話說 此系統可以很輕鬆的支援到數十個以上的 clients ，因為傳輸單元，消耗的運算資源很低;規劃單元雖然會隨著 clients 的數目而相對的增加，但其運算量也相當的低，所以不會影響系統的整體表現.所以關鍵其實是在於壓縮轉碼單元.

根據反覆的量測，在此系統裡， 壓縮一張 320x240 大小的全彩

影像約需要花費 7 msec 的時間.以 frame rate 為每秒 25 張 相當於 famer 間隔時間 40 msec 為例，在這 40msec 之內，大概要花 10ms 做轉碼，傳送一組封包(約 5~6 個)給一個 client 要花 1ms.所以此系統可以支援到 30 個 clients.若要同時傳送三個不同的即時影像，光轉碼和解壓縮再壓縮成 Mpeg-4 就會花掉 40 msec 裡的 30 msec 的時間，所以能個服務的 Client 就減少剩下 10 個.

如果利用無線傳輸，則傳輸的時間會增加.以 802.11 無線網路為例， 再用 802.11g 規格之 AP(Access Point)負責收/送無線數據資料.系統最大負荷量約為同時傳送已經儲存好的資料給 6~8 個 client，這是實際測量的結果，在 5.4 節我們還會有更詳細的分析.若要傳送即時的影像，則還要多花 10ms 做壓縮的動作，則最多只能服務 4~6 個 client.

(3)即時影像接收程式(客戶端程式):(Client)

任何有安裝 Client 程式的 Window 電腦，PDA，手機，都能拿來當作影像接收機.客戶端程式已經加了一套簡易的影像處理程式，可以做手勢辨視，分辨出病人比出什麼手勢，程式再跟據手勢做出回應，例如發出警報訊號給護理人員.另外還做了一個移動物體的定位系統，可以得知是否有人進入某個區域，例如說有外人入侵，或是病

患離開了病房. Client 可以很簡單的掛上外加的影像處理，以配合不同場合的應用.

以 Client 端而言，大致上可分為三部分，分為資料接收與重組模組、解碼模組像處理模組，三者的程式互動關係決定著 Client 端的系統穩定度問題。

以整個系統面來考量，資料接收與重組模組需要將 Server 端傳輸過來的 RTP 封包順利的接收到 Buffer 裡，然後再將 Buffer 內的 RTP 封包重組成 MPEG-4 格式的 Frame，為了使後續處理程序順利的執行，此模組需要快速的將接收到的 RTP 封包擺放到 Buffer 裡並快速的進行 MPEG-4 Frame 的重組動作。針對解碼模組部分，此模組的工作為將 MPEG-4 Frame 解碼成 RGB format 的影像。針對影像處理模組部分，此部分的主要工作為針對 RGB 影像進行相關的影像處理，所花費的時間取決於影像處理使用之演算法的複雜度。上述三者的運作順利與否，決定著 Client 程式的系統穩定度。

在此平台下，三個模組的實現是使用 thread 及 function 的程式所實現的，各模組為了完成它們的工作，皆需要花費某些時間予以完成。以此平台為例，Server 端每秒傳送 25 個 Frame 給 Client 端，Client 端要在 1 秒的時間裡處理這 25 個 Frame 的話，每個 Frame 之到達間隔為 40ms。然而資料接收與重組模組、解碼模組所花費的系

統時間約為 2 毫秒左右，而影像處理模組所花費時間則取決於演算法複雜度。影像處理模組主宰了 Client 端系統的穩定度，因此我們要試著解決影像處理模組所造成的影響，解決方法是使用程式的相關技術予以改善。

因為三模組是使用 thread 的方式所實現的，因此可設定各 thread 的優先權來改善三個 thread 間所分配到的執行時間。方法如下：將需要花費大量運算時間的影像處理模組設定為低優先權的 thread，需少量運算時間的資料接收與重組模組、解碼模組設定為高優先權的 thread。Thread 的優先權設定完後，OS 會給高優先權的 thread 較高的執行頻率，低優先權的 thread 則給予較低的執行頻率。

也就是說資料接收與重組模組及解碼模組都可以配合傳送端的傳輸速率來接收傳送過來的影像並順利的顯示在接收端，影像處理模組則根據影像處理所需要的規格來抓取影像。

系統應用

我們相信影音訊息即時傳送系統在應用上可以有很大的發揮空間，圖 5.3 是較詳細的系統架構圖:

圖中間下面的藍色框框即是我們的 Server，負責蒐集和儲存左方紫色框框內的影音資訊擷取系統傳來的資料，並且根據右方橘色框框

內用戶端要求的服務而送出即時或事前儲存的資料.

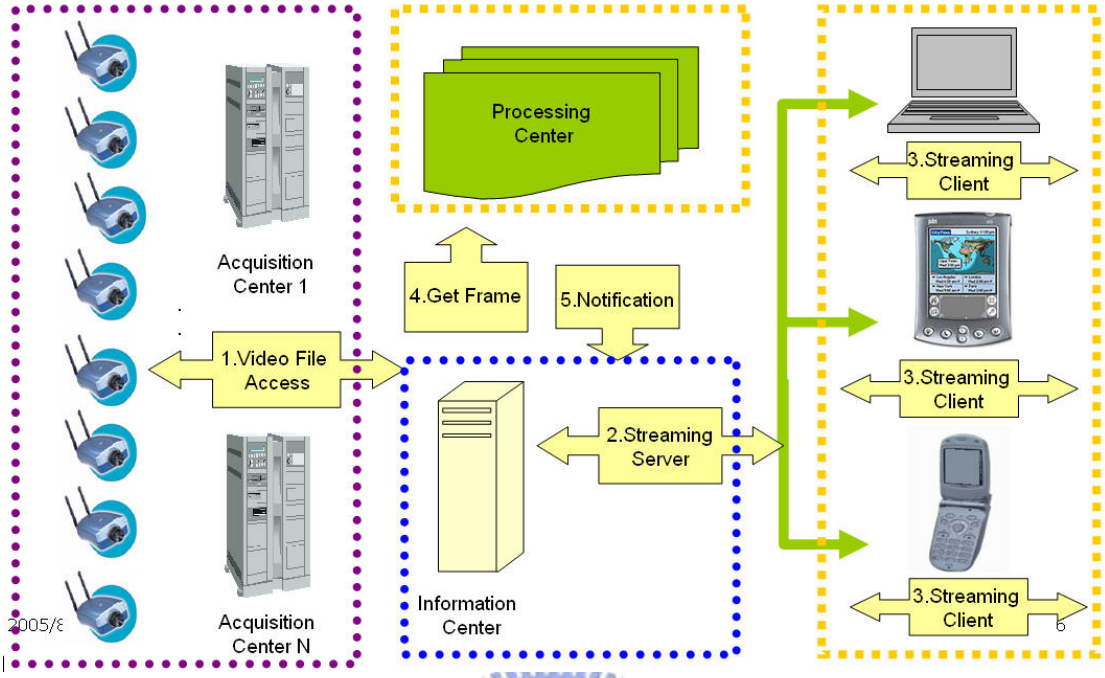


圖 5.3 系統架構圖

我們的 Server 可看成是個資訊收集中心，不管是聲音影像，或是各式各樣的訊號，只要能透過適當的界面擷取到 Server 程式內部，就能儲存或即時傳送給 Client.

目前程式已能抓到由網路卡(有線或無線)及以 USB 界面傳入的訊息，也將更進一步延伸到 RS232，IEEE1394 等其它常用的介面.

5.2 系統實作

目前傳送 stored video 已有做 smoothing. 但在傳送攝影機即時影像的情況(也就是 online 資料傳輸)並未加上 smoothing.所以現在

的目標就是將此演算法實作上去.

首先，先了解 Server 如何傳送即時影像.攝影機每秒鐘抓 25 張照片，每張照片用 JPEG 壓縮後經由無線網路傳到 Server，在 Server 內會將 JPEG 解壓縮，再從新壓縮成 MPEG-4 格式的資料，存在 Server 的 buffer 內，然後再切成一個一個的封包送出去.這個流程就是下面的系統架構圖中的左半部在做的事情.

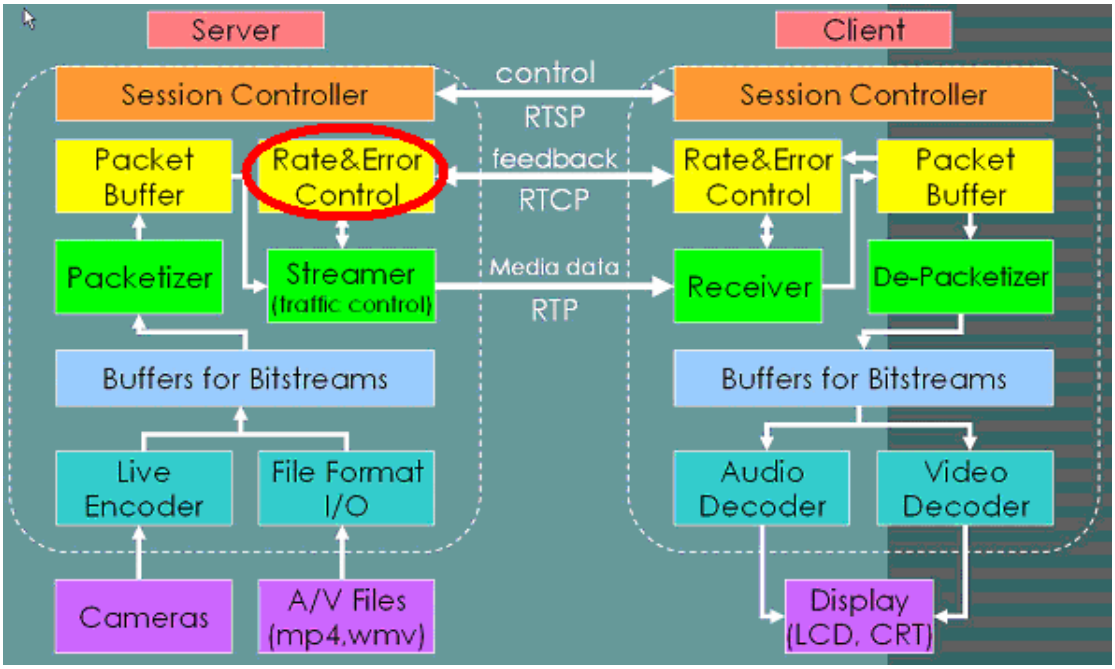


圖 5.4 控制封包傳送速度的單元在整個系統中的位置

我們之前已經完成的 stored video smoothing 是在 rate & error control 這個方塊中實作的，在圖中以紅色圈圈標記出來.在此方塊裡有個 thread 叫作 smoother，當有一個 client 連進 server 就會啟動一個 smoother，它不斷看系統時間，每經過 40ms，就會發出數個 token 給 streamer，而 streamer 拿到一個 token 就能丟出一個封包，

基本上就是藉由發 token 的快慢來掌控系統的傳輸速度.所以我們只要在開啟 stored video 檔案時，抓出這個影片的 video trace，用 MVBA 運算一次就能得到 smoothing 的排程，接著把這個排程經過 packet base 的修正，最後修正後的排程會送到 smoother 的手上，smoother 就是根據這個排程來決定每 40ms 要丟出幾個 token.例如說，smoothing 的排程從第一個 frame 開始依序是 6，5，5，6，5，4，5...個封包，則 smoother 這個 thread 就會第一個 frame 時間放出 6 個 token,過了 40ms 後的第二個 frame 時間,就看表放出 5 個 token,再過 40ms 放出 5 個 token，再來依序是 6，5，4，5...

我們發現在 online smoothing 的情況，從 camera 來的資料會存在一個 buffer 裡，Buffer 是一串 Link List，如圖 5.4:

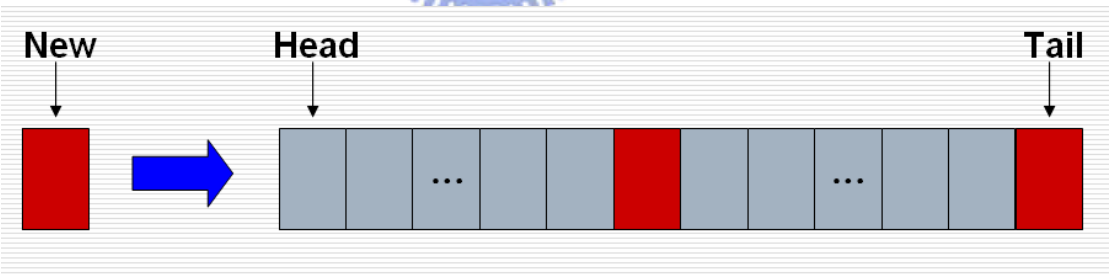


圖 5.5 online 影像緩衝區之結構

每個 node 代表一個 frame,新進來的 frame 會接在 Head 的前面.當新進來的 frame 是第 3 個 I frame,那 server 就會把第 2 個 I frame 之前的 frame 從 buffer 中清掉.這樣設計的目的在於，讓 client 抓的第一張 frame 是 I frame.因為 client 剛連進 Server 時，總是從

link list 的尾端往回抓資料，而 link list 的尾端會保持是 I frame，所以 client 抓的第一張 frame 會是 I frame.另外有一種情況，如果有個 client 還沒抓完從第一個 I frame 到第二個 I frame 之間的所有資料,在 Server 端的 buffer 會因為拿到了第 3 個 I frame 而清掉了第一和第二個 I frame 之間的資料，那 client 就會跳過那些被刪掉的 frame，直接從下一個 I frame 開始抓取.至於為何會有 Client 還在抓資料，Server 卻把那些資料清掉的情形，可能是因為 Client 進來抓第一筆資料的時間點是 Server 快要拿到第 3 個 Frame 的時候，或是 Client 抓到一半就當機或是遇上突如其來的網路延遲，使得 Client 從 Server 抓取 Frame 的速度沒辦法跟上 Server 從 Camera 抓 Frame 的速度，都有可能會造成上述的情況，但在穩定的情況下，這種情形是不會發生的.

總之，buffer 內最多只會有 2 個 I frame. 當然，我們可以自行調整 buffer 內最多可容許的 I frame 數目.若這個參數愈大，代表 Server 的 buffer 容量愈大,我們能夠做 smoothing 的視野就愈大(也就是前面提到的 window size， W)，做出的平順排程可以更平順，但是相對的，client 會感覺到更大的延遲.這個參數設成 2，就是代表即時性很高的 online 傳輸.

那麼，在 online smoothing 的情況，我們要怎麼修改現有的

thread 呢?online 與 stored video 最大的差別是資料的來源，從檔案變成記憶體中的一個區塊.我們想要把第 3 章的演算法套到傳輸平台中.在演算法中有幾個重要的參數: W: window size， alpha: 運算周期， b: client 端 buffer 大小，還有就是 d(t): 在 window 內每個 frame 的資料量大小.

我們需要的參數都在手邊了，現在只要在 streamer 抓取 buffer 裡 online 的資料時，讓 smoother 這個 thread 每 alpha 個 frame 時間長度做一次 packet-based MVBA 運算，就能決定從運算的時間點起，到下 alpha 個 frame 時間的 smoothing 排程.所以需要對現有之 stored video smoother 的修正，就是加上一個能即時從系統中得到 MVBA 運算所需各項參數的涵式，以及每 alpha 個 frame 時間長度就運算一次的判斷式，然後在 streamer 中找到 online 的流程，在適當的地方放上一個讓 smoother 插手管理丟封包的機制.windows 作業系統提供了一組 post()和 pend()指令，當執行到 pend()這行指令時，若記數器為 0，程式會停住，等到其它 thread 執行一個 post()，計數器內的數字+1，pend()被鎖住的地方就能繼續跑(計數器內數字會-1).我們利用這組指令來完成 smoother 的控管動作.用 pend()讓系統在丟封包前停下來，而 smoother 是另一個 thread，看系統時間以及 MVBA 運算結果執行 post()，這個 40ms 的 frame 要 5 個封包的話，

就執行 5 次 post()，系統中負責丟封包的 thread 就可以丟出 5 個封包，要丟第 6 個封包前會被 pend()鎖住，在 40ms 後等 smoother 執行了 post()，才能繼續丟封包.

所以 smoother 是一個獨立於傳輸 thread 外的 thread，負責計算 smoothing algorithm，並且每隔一段時間跟據 smoothing 的運算結果送出 post(). 負責傳輸的 thread 在 semaphor 拿到 post()後就能繼續動作，一個 post()可以讓 streamer()送出一個封包.

5.3 在無線環境中多用戶使用系統的情形:

我們建構的傳輸平台能讓多個 client 同時存取 Server 的資料，在傳輸平台同時服務多個用戶的情形下，我們想觀察封包在經過網路傳給用戶端後，smoothing 的效能是否會受網路的影響，我們列出四個觀測項目:

1. 在 client 能看到從 server 傳過來的圖片，圖片如何經由無線環境傳送的?在這段傳輸過程中的延遲及 jitter 有多大?
2. 觀察 client 和 server 間以無線網路傳輸資料時，client 數量增加對 smoothing 演算法的影響.還有就是 client 數量多的時候，server 傳送封包的順序有什麼變化?且 server 傳出的資料量結合了要送給不同 client 的資料，相當於是很多亂數的相加，我們觀

察 server 端的輸出是否不靠 smoothing 本身就很平順了.

3. 比較在傳輸過程中，傳送端送出封包 和 接收端接收封包的情形，觀察平順演算法的排程在經過網路的延遲和 jitter 之後，會變成什麼樣子.
4. 承接第 3 點，進一步在 MAC 層觀察 802.11 機制的運作，以了解封包延遲的可能原因，並推算出在 802.11 無線網路下，系統能負荷的 client 數量.

觀測工具:

(1)WinPcap: 一個可以抓取通過網路卡的封包的函式庫.在 Server 和 Client 端各用一個 WinPcap 來記錄通過的封包的時間及大小.我們在 Server 端只記錄目的 IP 位址是 Client 的封包,在 client 端只記錄來源 IP 位址是 Server 的封包.

(2)Airopeek: 一個可以聽取並分析 802.11 無線訊號的應用軟體.

我們另外設置一台 NoteBook，在上面跑網路監聽程式，抓取所有的無線訊號.

觀測方法:

在一 PC 上分別開啟 server 程式,在經由無線網路連結的另一 PC 上開啟 client 程式傳送 cindy.mp4 短片 ，在 streaming 傳輸的過程

中，分別在傳送端和接收端測量進/出封包的大小和到達/離開之時間， 這些封包只包括以 server IP 為 source 及 client IP 為 destination 的封包，共記錄 5000 個封包，並以此測量結果計算出隨時間經過累積的封包量和傳輸速率之變化. (圖)是我們測量的環境和使用的工具:(箭號表示封包的流向)

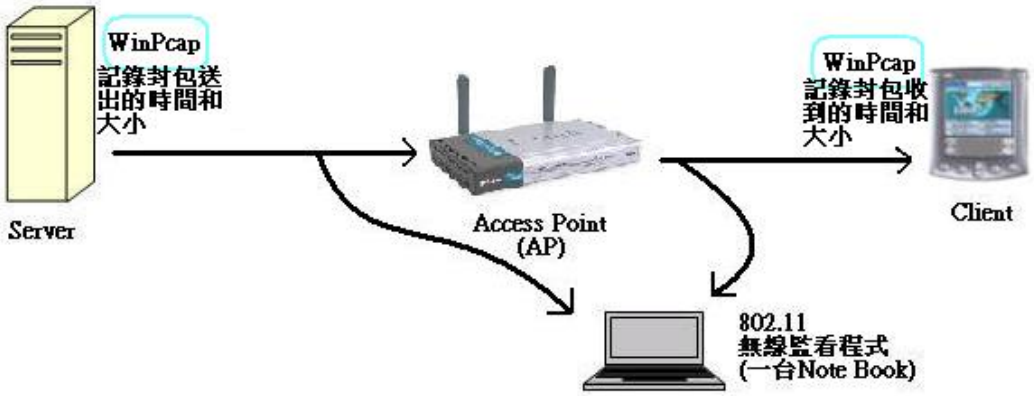


圖 5.6 測量的環境和使用的工具

從 server 丟出來的照片，切分成數個封包後，送到網路卡，硬體將數位訊號變成無線電波打到空氣中，封包的訊號被 AP 收到後，再從 AP 送到 client.

接著我們分成三個小節，針對前面提到的 1~3 個項目，說明我們觀測到的結果:

5.3.1 網路延遲及 jitter 之量測

5.3.1.1 網路延遲

我們用"ping"指令測量網路延遲，在沒有人使用的時候，ping 值約在 1~2ms 左右，大概 10 次裡會有 1 次跳到 100ms 以上. 而在網路忙碌的時候(例如說，有 5 個 client 連上 server 的情況)，ping 值就提高到 30~100ms 之間，且不時會破千及 time out 的情況. 網路延遲基本上是和網路的忙碌程度最有關係，當然傳輸途徑影響也很大，不過我們為了不受到外部網路的影響，只在實驗室內部的無線網路做量測，所以我們用的傳輸路徑都是一樣的.

5.3.1.2 網路 jitter

如果網路延遲的時間一直都不變，那麼 Server 每隔 40ms 丟出一張 frame 的資料，Client 在經過一小段網路延遲後，也會每隔 40ms 收到一張 frame 的資料，我們做個實驗，來看看實際上的情況是否如此.

首先，把接收端收到封包的時間加以分析，大致上封包是一群一群地到達，也就是 4~7 個封包組成一個 frame，每 40ms 就來一張 frame. 如圖 5.7 所示:

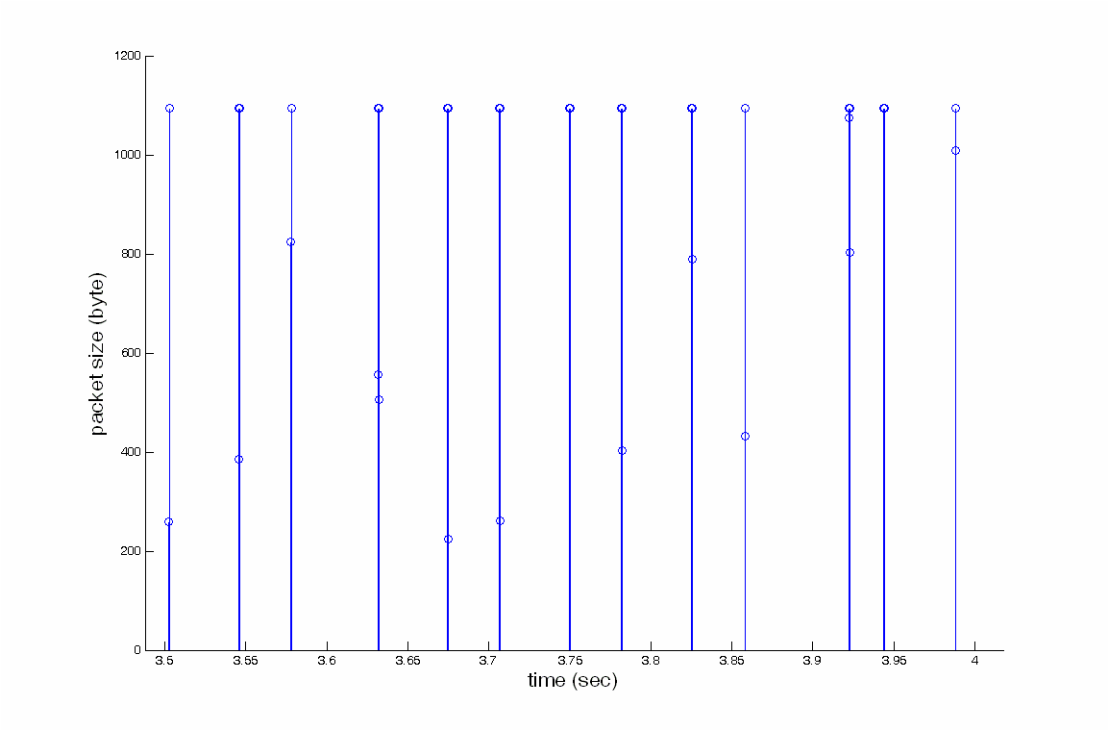


圖 5.7 封包到達之相對時間

圖中橫軸是時間，封包群的到達時間間隔看起來有些變動.我們做個統計，時間間隔數目統計如圖 5.8:

由於是在接收端量的，加進了網路延遲的影響，看起來有點亂，frame 和 frame 間的間隔從 10ms 到 60ms 都有;我們比較一下傳送端看到的情況，如圖 5.9:

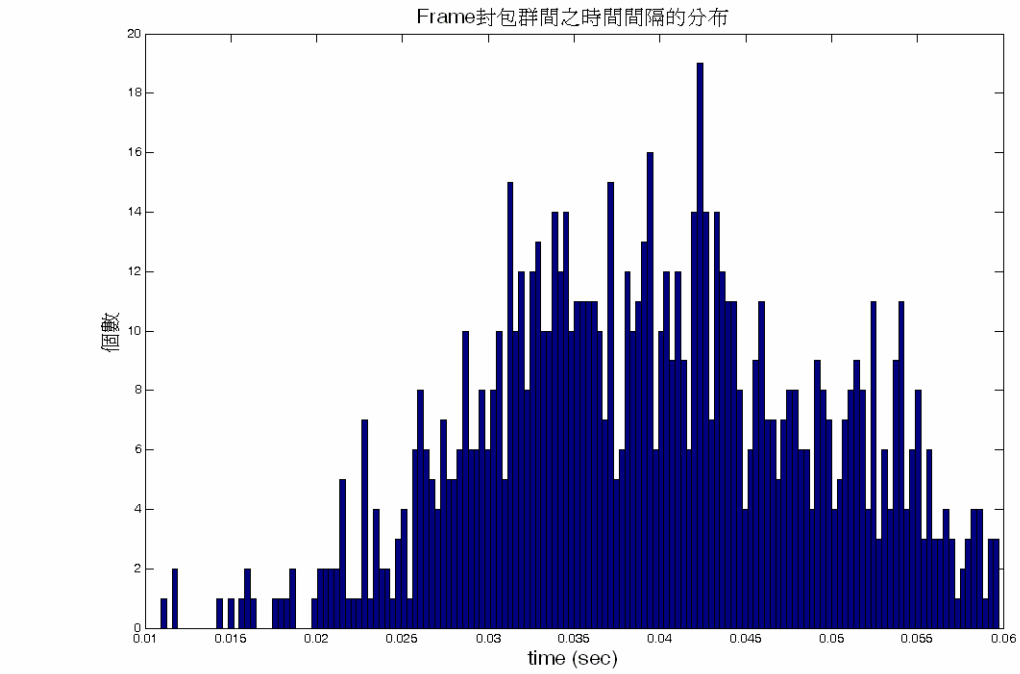


圖 5.8　frame 間隔時間的數量統計（於接收端量測）

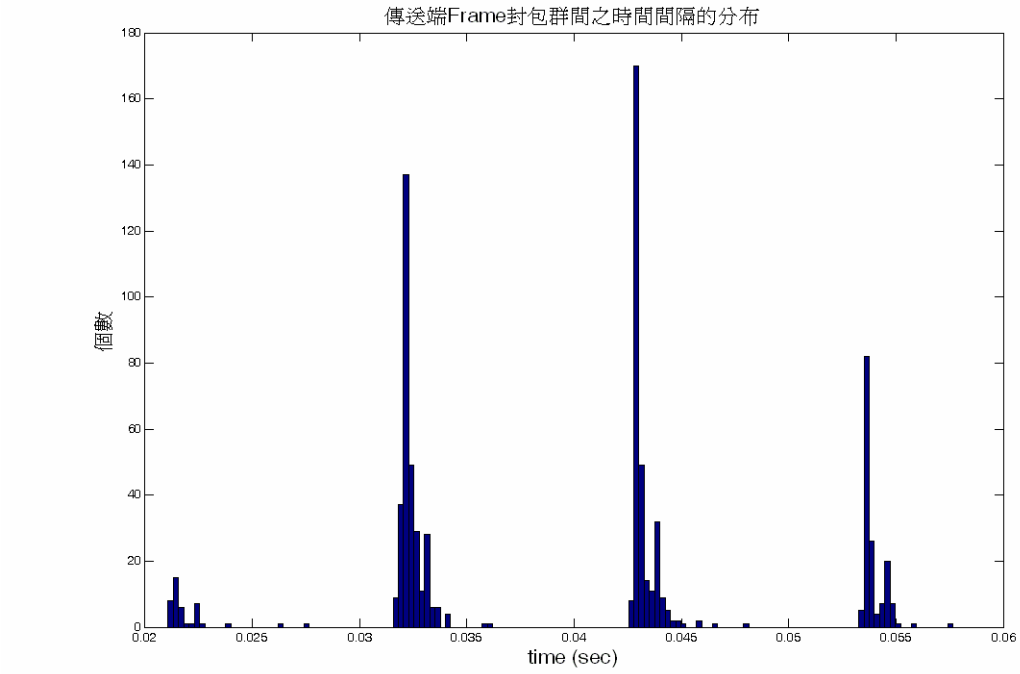


圖 5.9 frame 間隔時間的數量統計(於傳送端量測)

可以看出集中在 22ms，33ms，43ms，54ms. 我們寫程式時是設計

每隔 40ms 送出一群封包，但在 WinPcap 量到的情況顯示，封包並沒有精準地照規劃每 40ms 就送出一群.

我們推測是因為程式有太多 thread 同時在跑，所以造成這種現象，所以另外用一個簡單的 client-server 程式來傳送封包，觀察情況有何不同，結果如圖 5.10:

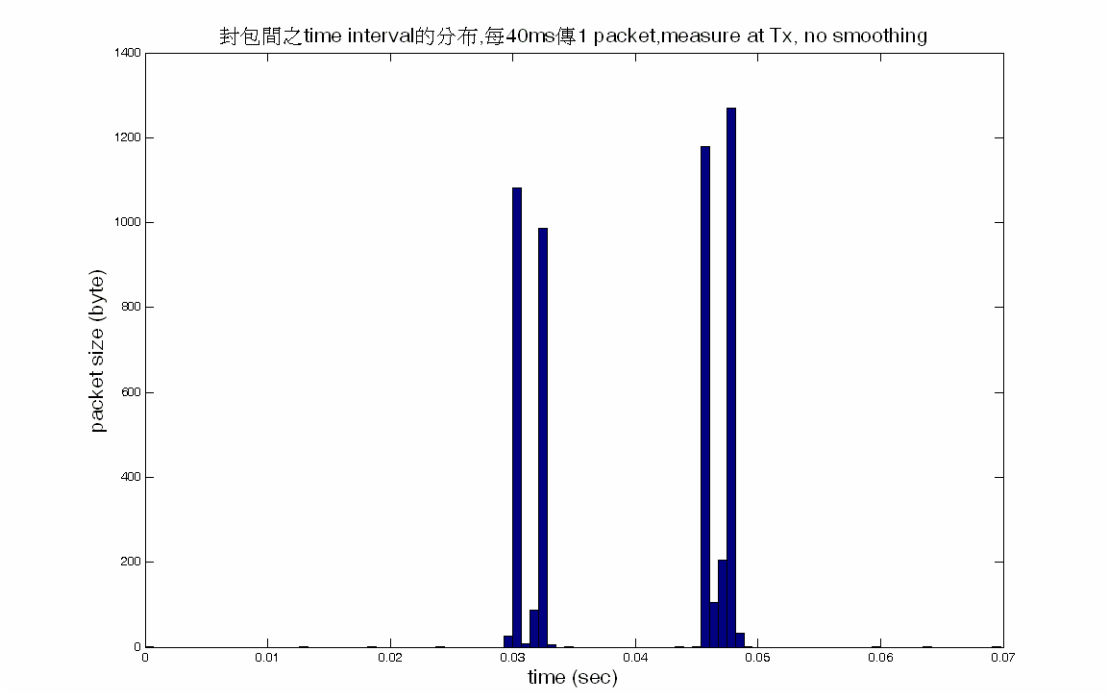


圖 5.10 frame 間隔時間的數量統計(簡單的 rtsp client-server 架構)

可以看出集中在 32ms，46ms. 比起在複雜的 client-server 架構下，分布單純了一點，但沒能準時在 40ms 送出的特性依然存在.這牽涉到作業系統如何管理各 thread 的運作，複雜性頗高.

5.3.1.3 Client 數量對平順演算法的影響

我們同時開啟 6 個 client 連上 server，量取 server 和某一個 client 的流量，如圖 5.11

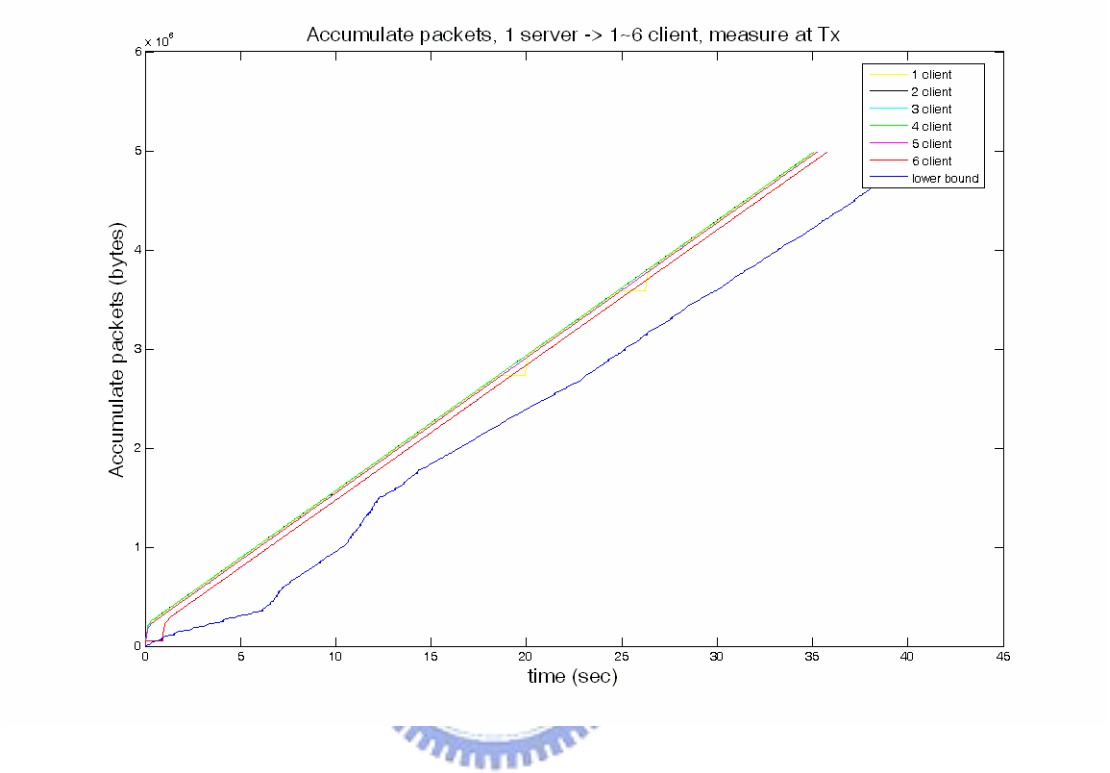


圖 5.11 在多用戶同時使用系統時，其中一用戶的時間-封包累積量

這是封包累積量對時間做的圖，最底下的藍線是傳輸的 Lower bound，緩衝區大小是 2Mbytes，在有做 smoothing 的情況，每 40ms 累積的封包量幾乎是固定的，所以斜率看起來沒太大變化，也就是傳輸速率變化很小.

我們將上圖在 0~1.2 秒附近的圖放大來看，如圖 5.12

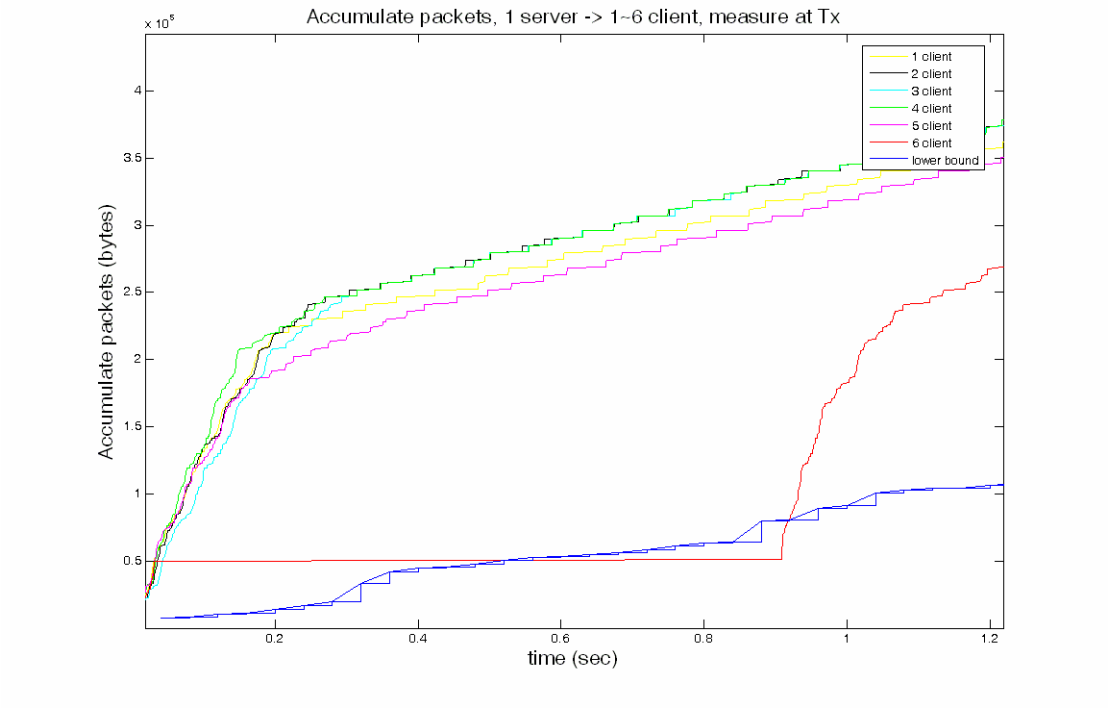


圖 5.12 時間-封包累積量（區間 0~1.2 秒）

由於是傳同一個檔案，所以理論上在同一時間所累積的量應該是一樣的，在 2 個 client，3 個 client，4 個 client 可以看出幾乎是重疊的，這是最合理想的狀況;但在 6 個 client 的情況，server 停了約 0.9 秒沒傳東西.有點像是當掉了.應該是 server 同時跑 6 個程式，太忙而造成的現象，另外，1 個 client 和 5 個 client 的曲線和 2，3，4 個 client 的差較遠，推測也是作業系統執行上的問題.

為了確定 6 個 client 發呆的情形不是偶發的現象，我們多做了幾次量測，發現 10 次中約有 1 次會發生 server 發呆的情形，接下來是另一次量測的結果.

因為封包累積圖需要在 MATLAB 下縮放移動才容易看，為了在文

章中能讓讀者方便了解，我們接下來用速率-時間取代封包累積-時間的表示法.然而，這樣的表示法會喪失一些時間上的資訊，因為求速率的時候，必須取一段時間，然後用這段時間內送出的封包總量除以時間長度，但是這樣就無從得知封包在這段時間裡的那個時間點到達.

圖 5.13 和圖 5.14 是有做平順演算法的 server 服務 1~6 個 client 時，在傳送端/接收端看到的速率變化.

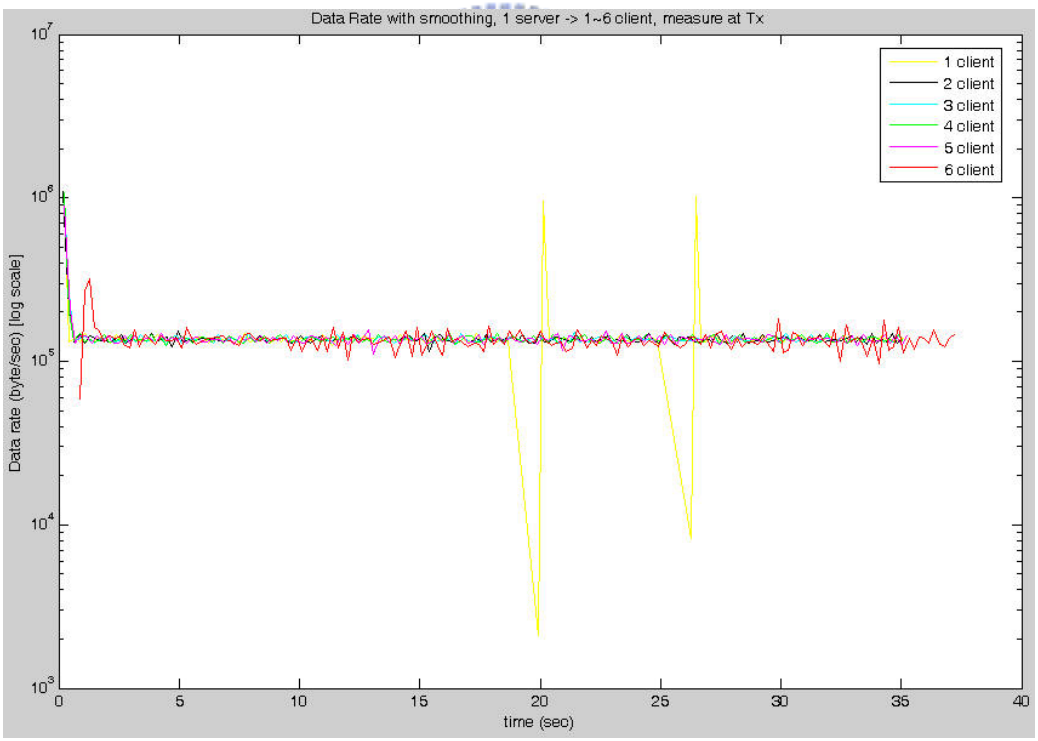


圖 5.13 傳送端送出封包的速率

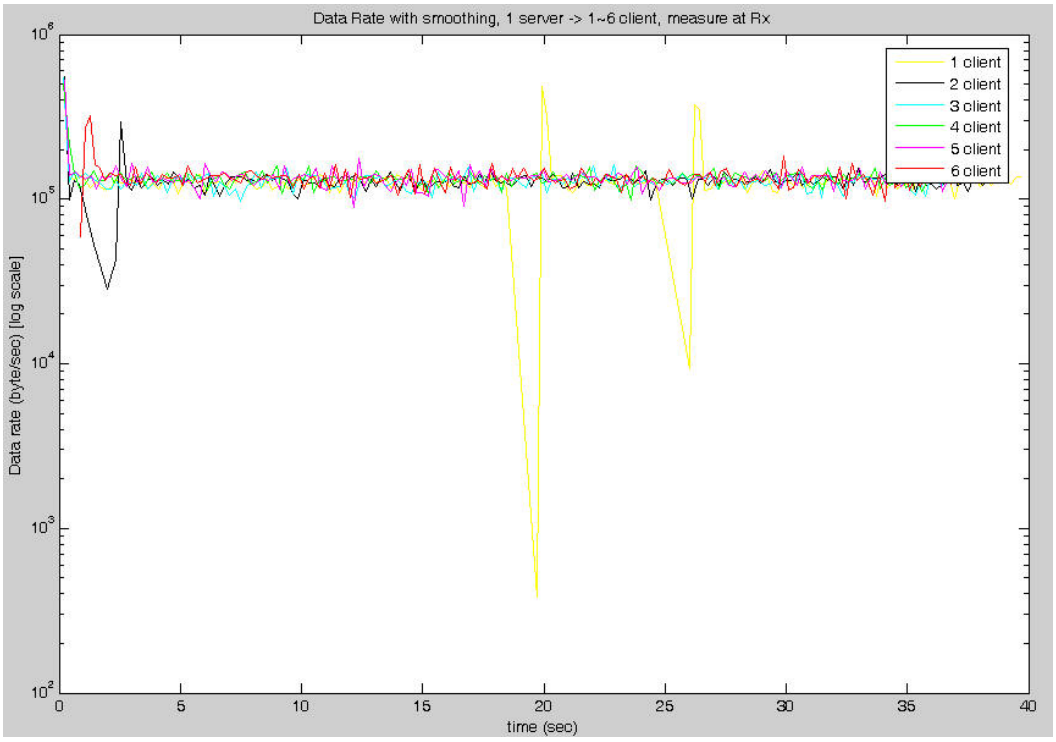


圖 5.14 接收端收到封包的速率

在 Tx 端的情況，可以看到 client 數量只有 1 的時候(黃線)，在時間 20 秒級 25 秒附近忽然陡降又回升，表示在這段時間傳輸的資料量很小，這樣的現象推測是程式 sleep 太久或是受到作業系統內其它程式的影響，也就是像(前圖)6 個 client 在 0~0.9 秒的情況;除此之外，client 數量對傳輸速率沒有很明顯的影響.這很合理，若 server 有足夠的運算能力，不管有幾個 client，每個 client 都會做 smoothing，彼此之間不會互相影響.

在 Rx 端的情況下，可以看到速率變動幅度比 Tx 端大，變化的情形和 client 數量也無關，Rx 端主要是跟著 Tx 端變化，再加上無線

網路的傳輸延遲.

圖 5.15~圖 5.17 是在 Tx 端比較有使用 smoothing 和沒有用 smoothing 的傳輸速率有何差異，因為不同 client 的圖都很像，我們挑幾個來看:

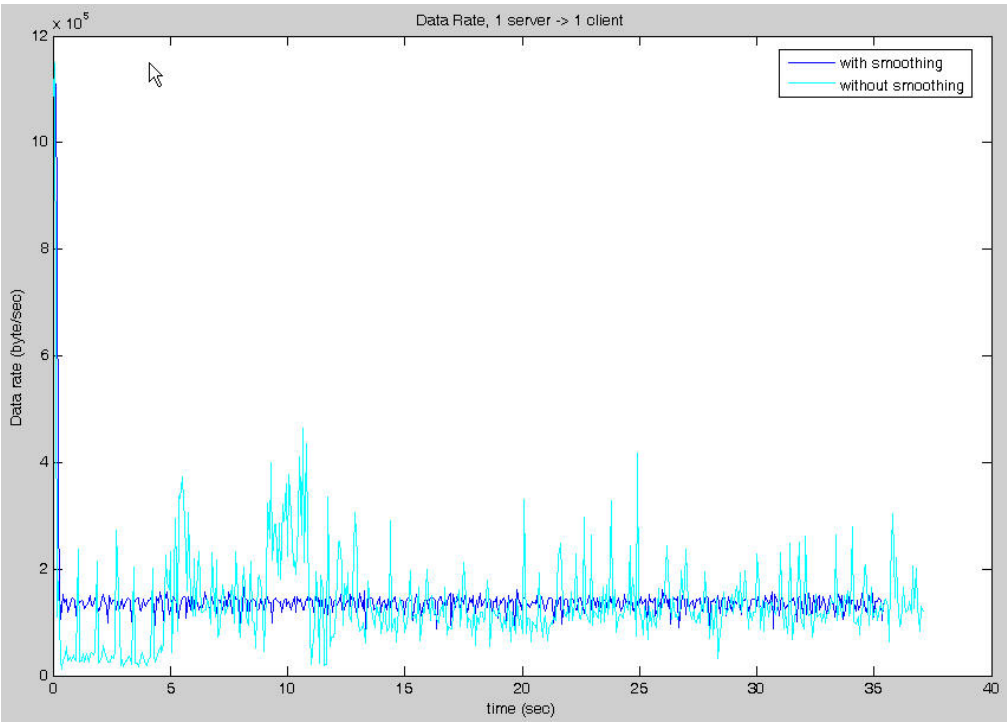


圖 5.15 1 client 之傳輸速率

圖中能看出有 smoothing 的傳輸速率(藍線)比沒用 smoothing 的(青線)平穩.在最開始的地方藍線衝到很高的地方，那是因為程式設計成在剛開始的時候要先送 25 個 frame 的資料量，好讓 client 的 buffer 能先裝滿，減少 underflow 的可能性.

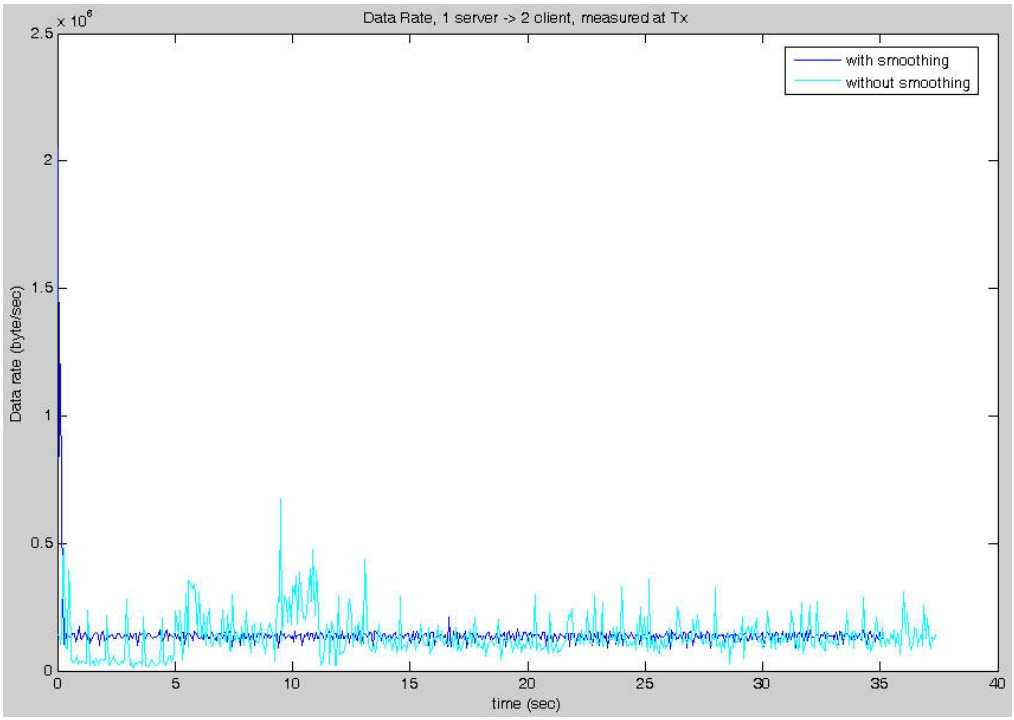


圖 5.16　同時有 2 個 clients 使用系統，其中 1 client 之傳輸速率

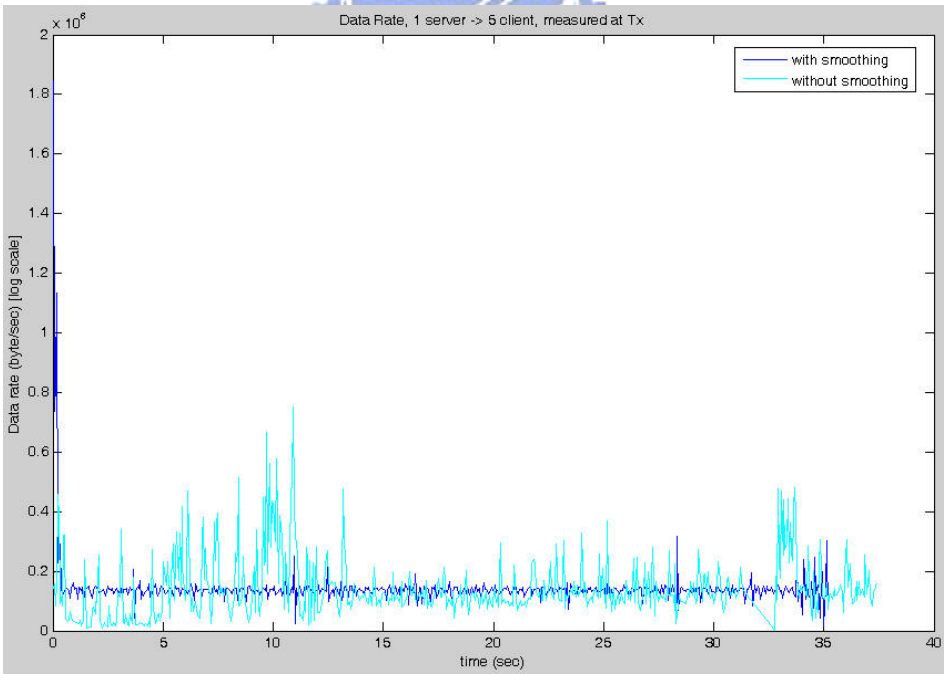


圖 5.17 同時有 5clients 使用系統，其中 1 client 之傳輸速率

再來，我們想知道有很多 client 同時連上 server 時，server 如何分配哪段時間傳送哪個 client 的資料. 圖 5.18 是 6 個 client 同時連上 server 時，每個 client 的封包大小-時間圖.

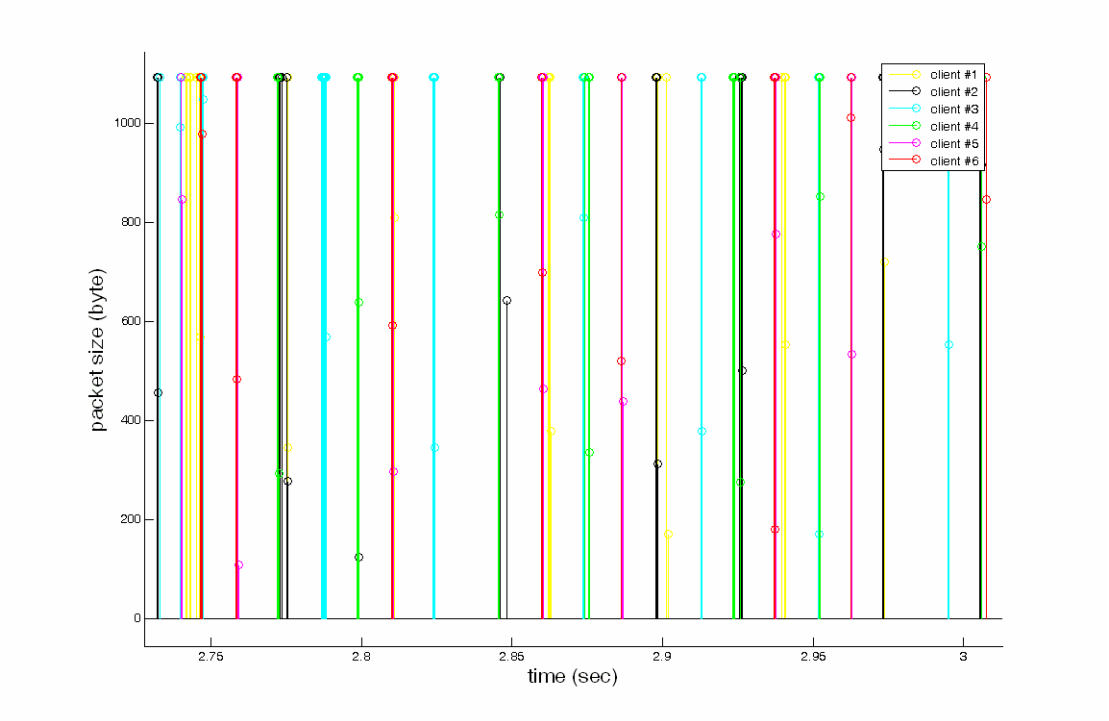


圖 5.18 clients 使用系統時的封包傳送相對時間

看一個 40ms 範圍內的傳送情況，如圖 5.19:

圖 5.19 中封包是一叢一叢的，代表一個 frame 包含的包，不同顏色表示不同 client 的封包. 在 server 端做規劃的時候是每個 client 分別獨立規劃的，不同 client 的封包群在時間上的距離有近有遠，看程式執行的時間;接著我們比較在接收端看到的情況，如圖 5.20:

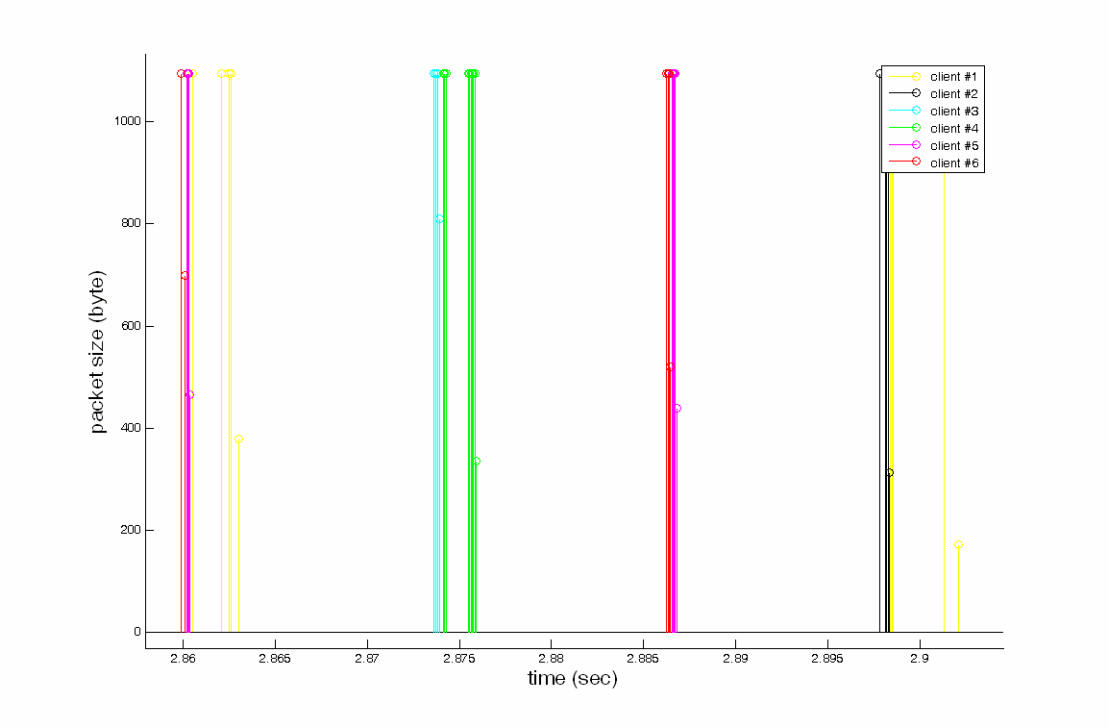


圖 5.19 有 6 clients 使用系統時的封包傳送相對時間(取 40ms 範圍)

圖 5.20 是 AP 傳給 6 個 client 的封包，可以看得出來封包間的距離比較開，因為封包實際傳送時要等 RTS/CTS，所以花得時間較多；另外可以發現，無線網路已經很忙碌，在時間軸上空白的部份(閒置)已不多，若再加上 Server 傳給 AP 的另外一半傳輸量，網路幾乎沒有閒置的時間了.

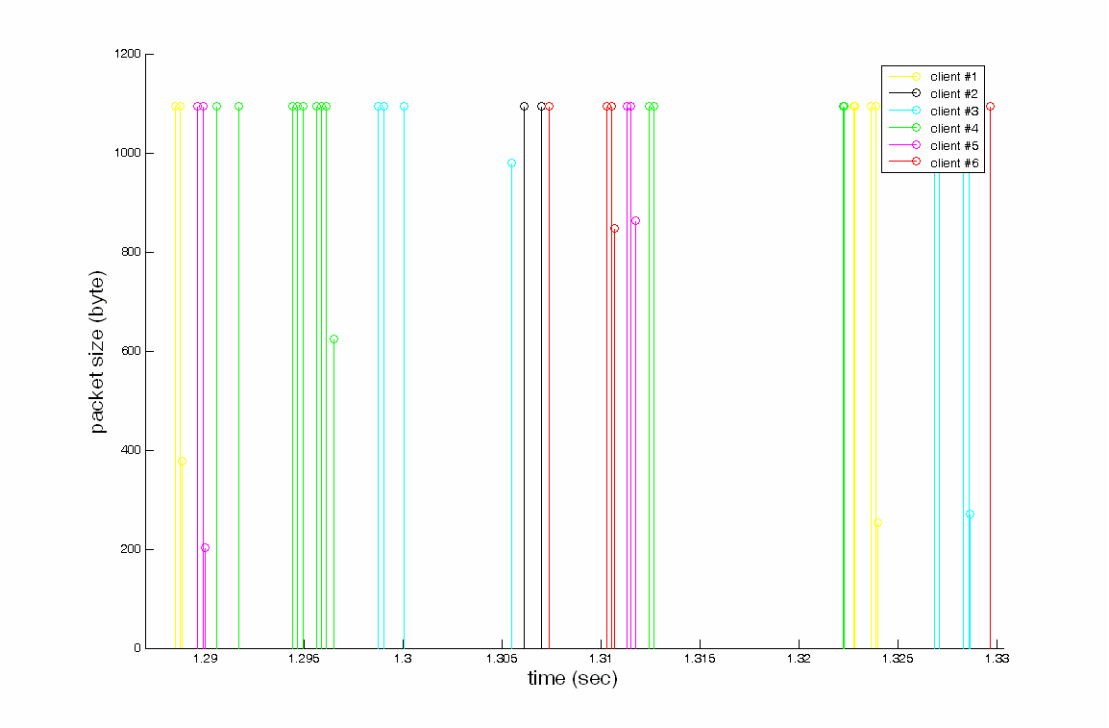


圖 5.20 有 6 clients 使用系統時的封包接收相對時間(取 40ms 範圍)

5.3.1.4 傳送端和接收端之比較

下面的圖是比較 Tx 和 Rx 兩邊的速率，6 張圖分別是 1 個有做 smoothing 的 server 連 1~6 個 client 的情況:

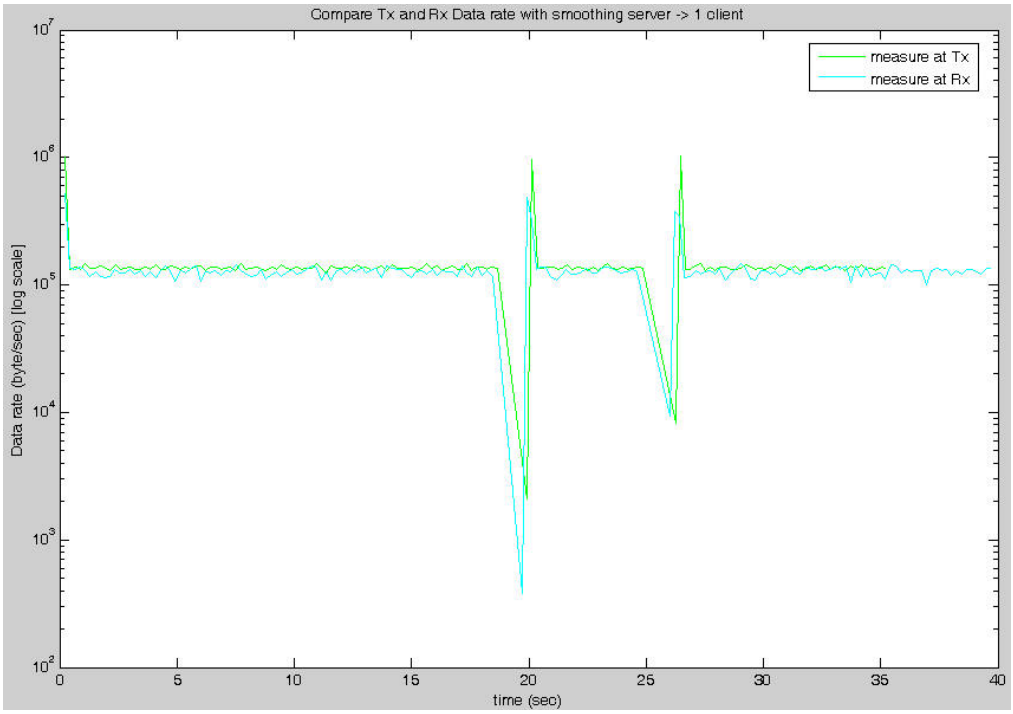


圖 5.21 單一 client 之傳輸速率，Tx/Rx 的比較

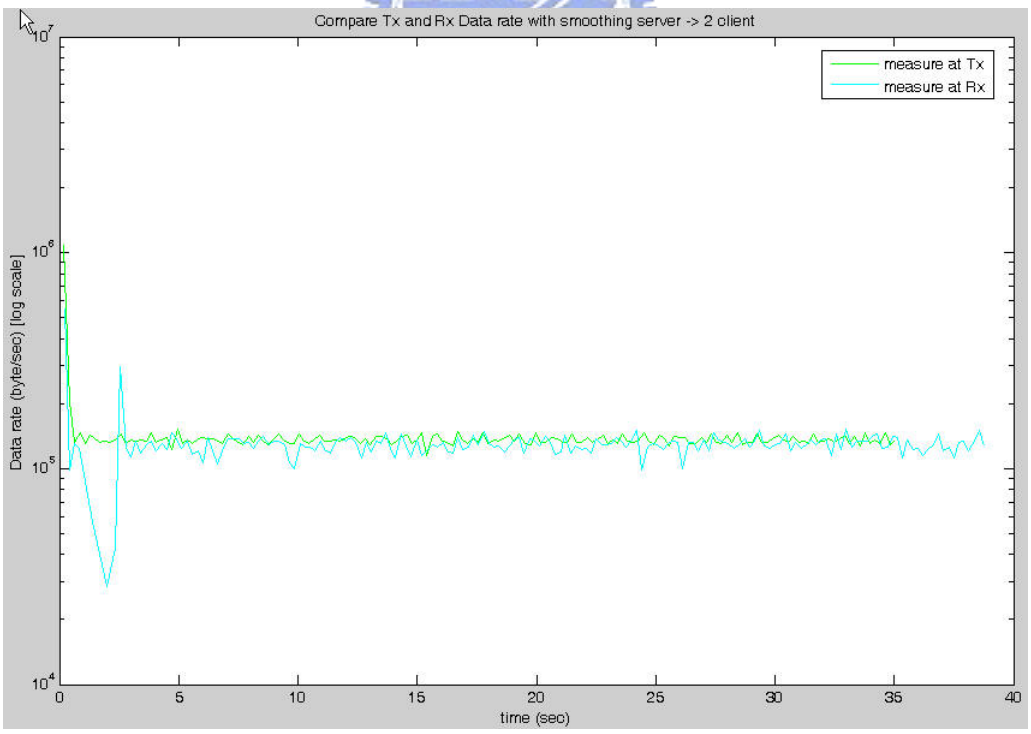


圖 5.22 同時 2 clients 使用系統, 其中 1 client 之傳輸速率, 在 Tx/Rx 的比較.

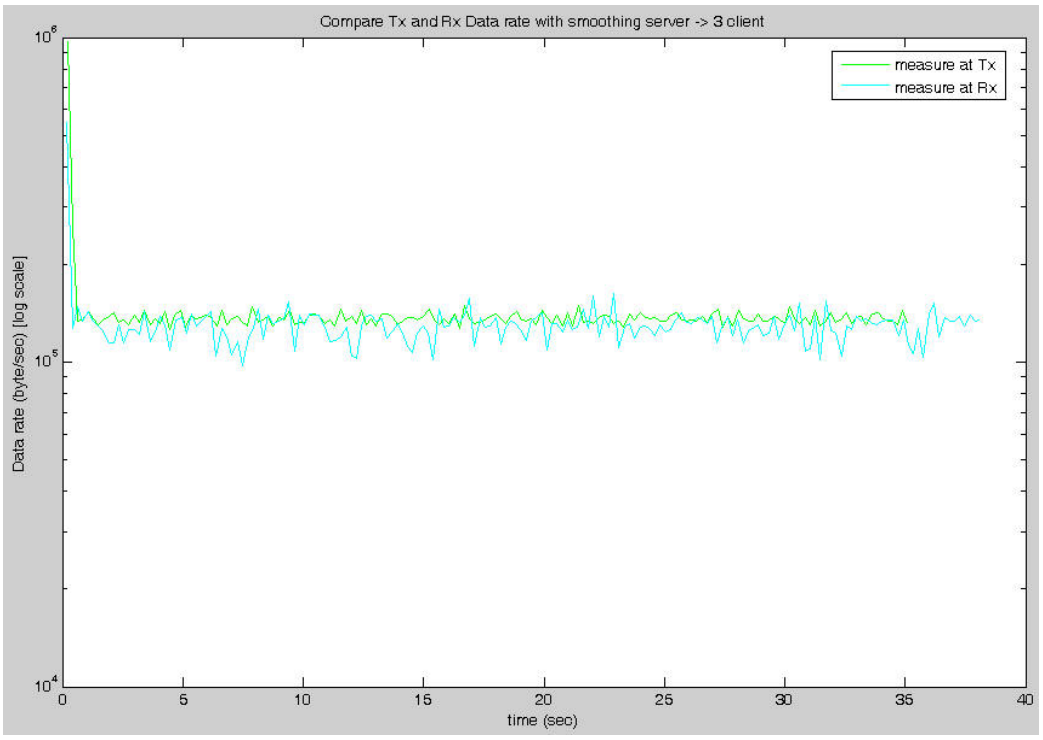


圖 5.23 同時 3 clients 使用系統，其中 1 client 之傳輸速率，Tx/Rx 的比較

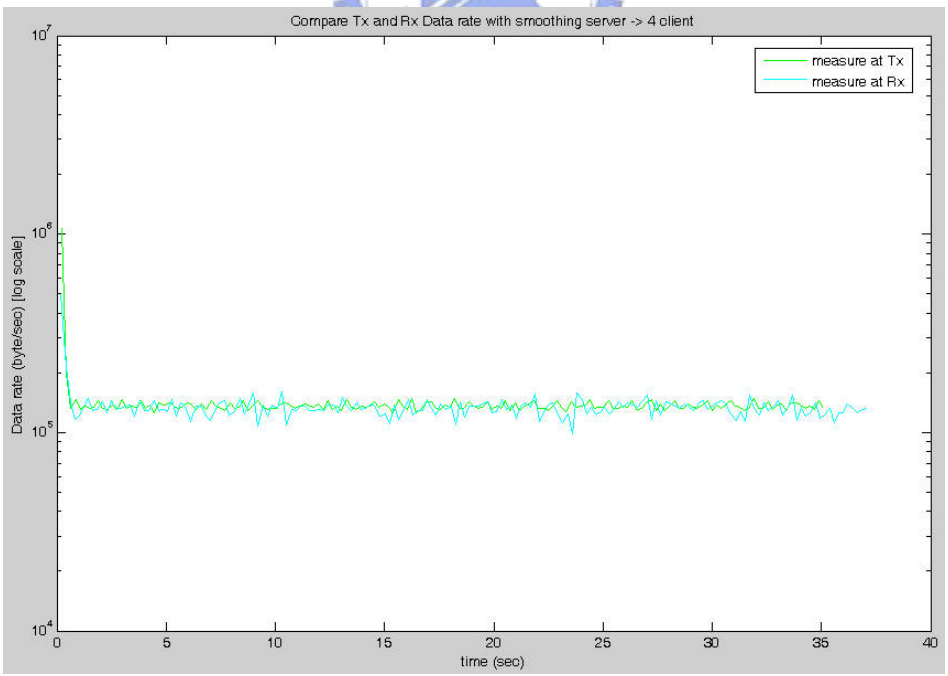


圖 5.24 同時 4 clients 使用系統，其中 1 client 之傳輸速率，Tx/Rx 的比較

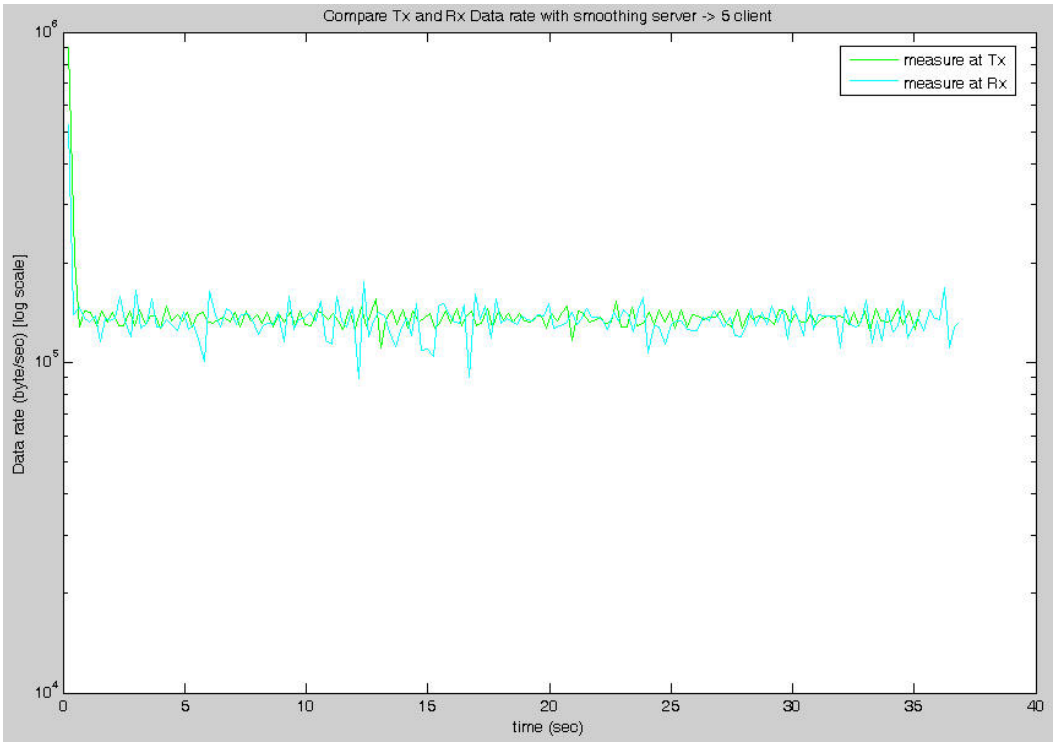


圖 5.25 同時 5 clients 使用系統，其中 1 client 之傳輸速率，Tx/Rx 的比較

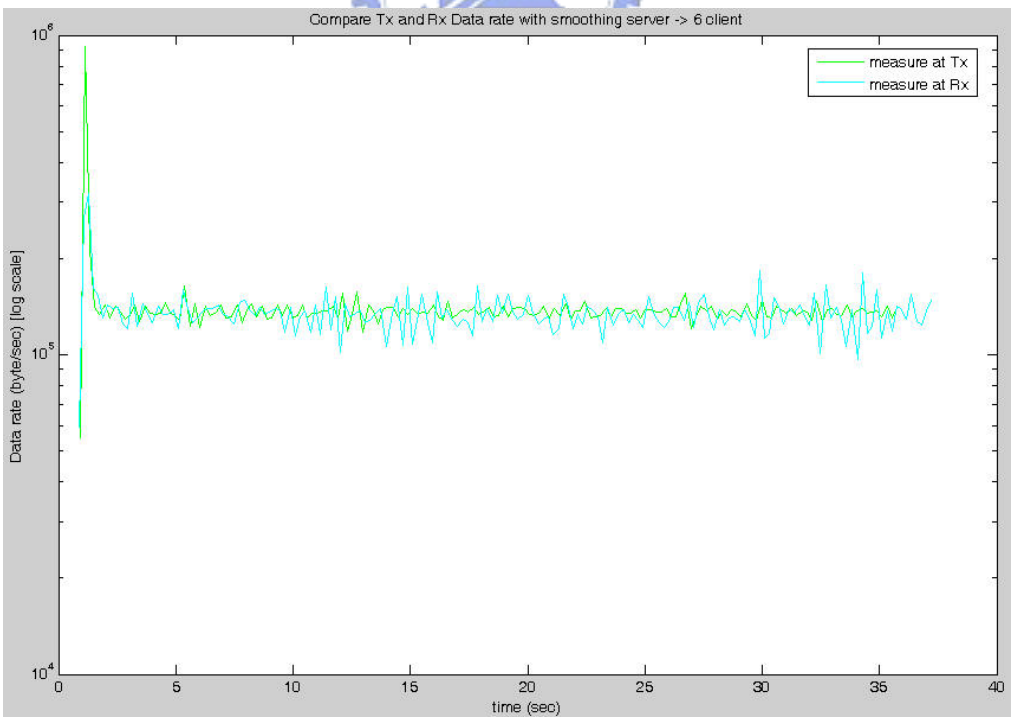


圖 6.26 同時 6 clients 使用系統，其中 1 client 之傳輸速率，Tx/Rx 的比較

從六張圖中發現的第一點是，接收端完成 5000 個封包的接收工作需要的時間比傳送端送出 5000 個封包的時間多了 2~4 秒不等.第一個可能的原因是接收速率比較慢，但後來我們又觀察到，負責轉送封包的 AP，從接收封包到傳送出去之間的延遲並不會長到 2~4 秒.所以這個解釋並不合理.經過比對特定封包，才發現是因為無線網路中封包遺失率較高，約有 5%~9%，導致接收端收到的第 5000 個封包是傳送端送出的第 5250~5450 個封包.而 cindy 這個 video trace 的平均封包約為 5 packets/frame，所以多出的 250~450 個封包約是 50~90 個 frame;以每秒 25 張 frame 的播放速度來計算，就是 2~3.6 秒，和我們觀測到的結果大致吻合.

我們在傳送端按照規劃好的傳輸排程切好封包，接著要取得無線網路存取權，才能把封包丟出去.在競爭無線網路使用權花掉的時間是網路 jitter 的成因之一，接著我們做個實驗來觀察規劃好的排程和實際傳到網路上的時間有何差異.

我們能在 server 內部用 WinPcap 程式看到封包通過 IP 層的時間和大小，接著送到網路卡發射出去後，在 Note Book 上的監聽程式能偵測到網路卡送給 AP 的封包. 我們針對同樣的連續數個封包(可以從 UDP 的 ID 找到相對應的封包)，比較在 server 端的 WinPcap 所看到的封包的時間間隔，以及在 AP 端(NB 的監聽程式)所看到的時間間

隔如表 5.1(單位 us)

編號	1	2	3	4	5	6	7	8	9	10	11
server	2270	14.7	90	80	53	53	140	53169	59	52	53
AP	868	3324	4590	1102	1287	46000	2171	46767	767	279	260

編號	12	13	14	15	16	17	18	19	20
server	52	32285	58	52	53	53	42485	63	51
AP	268	34148	268	270	1988	268	36672	621	85

表 5.1 封包間的時間間隔（server/AP 之比較）

從 server 排程好，到網路卡打出訊號，這中間的 jitter 最大是在編號 6 發生的，有到 46ms 之譜.推測是封包遺失或是 AP 太過忙碌的緣故.

AP 收到 server 網路卡送出的封包後，接著會轉送給 client，通常 AP 一次收完 5~10 個封包後，再一次丟給 client，如(圖)所示:

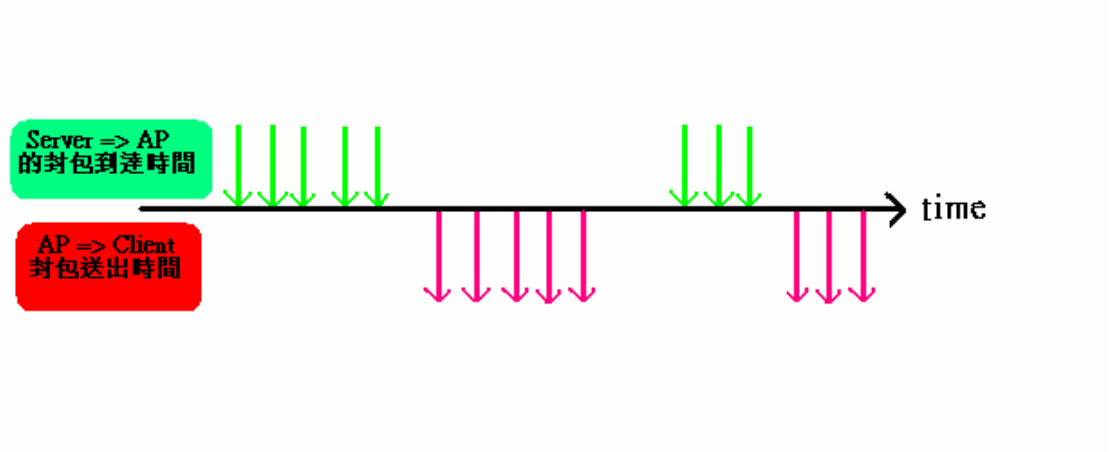


圖 5.27AP 收/送封包的情形

而 AP 送給 client 的封包，在監聽的 NB 也同樣能偵測到，我們比較同樣的連續數個封包，在 AP 端(NB 的監聽程式)所看到的封包的時間間隔,以及在 client 端的 WinPcap 所看到的時間間隔,如(表)(單位 us)

NO	1	2	3	4
AP	397	1510	43034	935
client	248	1661	43056	816

表 5.2 封包間的時間間隔 (AP/client 之比較)

觀測數據雖不多，但可發現兩者相差都在 200us 以下.這也很合理，理論上電波到達 NB 和 client 的時間差會在 33ns 以內，但兩邊的監聽程式不同，CPU 運算速度也不同，所以會有幾百 us 的差異.

再來探討傳輸速率和 client 數量之間的關係: 直觀地來想，若沒有其它人存取無線網路，那 client 數量愈大，每個 client 能分到的頻寬就會愈小，也就是 Rx 看到的接收速率會變小;但是在我們的測量結果中，Rx 看到的接收速率和 client 的數量間並沒有很明顯的關係.幾乎都是維持在一個固定的值(和傳送端送出封包的速率一樣). 推測其原因是在於，直到 6 個 client，都還沒完全用滿無線網路的頻寬，所以還不會有明顯的影響;然而在 client 到 7 個以上的時候，常常會發生 client 連結不到 server 的情況，所以最大 client 數目的量測工作不容易準確地判定.

5.3.1.5 無線網路之寬效率及系統能負荷的 client 數量上限

我們的無線網路使用 802.11g 的 AP 和無線網卡， 傳輸速率能到 54Mbps， 而和一個 client 之間的 streaming 傳輸要用掉 130Kbps 左右的頻寬，理論上能同時服務 400 多個用戶.

但在實際測量時，開啟一個 server，當我們慢慢增加 client 數量的時候，會發現 client 數量到了 6 以上之後，再開啟新的 client 時，連不上 server 的機率會增加，或是連上 server 之後，會停在 ”pause” 的狀態. 到了 8~9 個 client 時這樣的情況又更加明顯. 在多次測量的過程中，最多有連過 10 個 client.

想要解釋這個原因，我們先要了解無線網路在 physical lair 是怎麼運作的，這在接下來的 5.4.1 小節會有簡單的說明;在 5.4.2 小節，我們會試著從 MAC 的角度來解釋，為什麼系統的負荷量會像前段敘述的那樣.

5.4 無線網路 802.11 MAC 運作機制

為了更清楚我們的系統是透過何種網路介質傳輸資料，在這章我們簡單地說明一下無線網路是如何運作的.此傳輸系統使用 802.11 無線網路，想要分析在此網路下，系統最多能負荷多少使用者，就要先明白使用者是如何分享無線網路這個介質.也就是 802.11 MAC (Medium Access Control)的機制

5.4.1 802.11 MAC (Medium Access Control)簡介

在同一個頻帶上，AP 一次只能和某一台 camera 或是 server(或著叫做 Station)通訊..否則同一時間有多台 Stations 搶著要和 AP 通訊，就會有 collision 的發生，而造成訊號干擾嚴重.為了避免此種情況發生，我們要有個機制，讓每台 Station 知道這個頻帶何時是可以用的，何時是有其它人使用中.

802.11 的 MAC 很類似 Ethernet 的 MAC.不過有一個差異是，Ethernet 是用 carrier sense multiple access / collision

detection(CSMA/CD)， 意即每個 Station 有封包要傳就丟出去，等到有 collision 發生再重傳; 而在頻寬很珍貴的無線環境中，等到有 collision 再重傳是很浪費頻寬的，所以 802.11 MAC 用的是 carrier sense multiple access / collision avoidance(CSMA/CA)，也就是儘量避免 collision 的發生，而詳細的運作機制在下面介紹.

5.4.1.1 介質使用權的競爭機制

802.11 也像 Ethernet 的 MAC，是分散式的系統，沒有一個專門做中央控制的設備，只要 Station 有東西想傳，就檢查 medium 是不是可用的，如果檢查後發現 medium 已經 idle 超過一段 DCF inter-frame space (DIFS)的時間，就馬上傳送封包出去; 如果 medium 有人用，就等到 medium 再次變成 idle 後，經過一個 DIFS 並跑完 back-off 程序後再傳送;所謂的 back-off 程序，是說當 medium 變成 idle 且經過 DIFS 後，會有一段叫做 back-off window(或是 contention window)的時段，這個時段被切成許多個 time slot，(每個 slot 的長度是 10us)，Station 會隨機選一個 slot，並在該 slot 試著去使用 medium，而所有的 Stations 中，誰選到最前面的 slot 就有權利使用 medium.下圖是 back-off window 的示意圖:

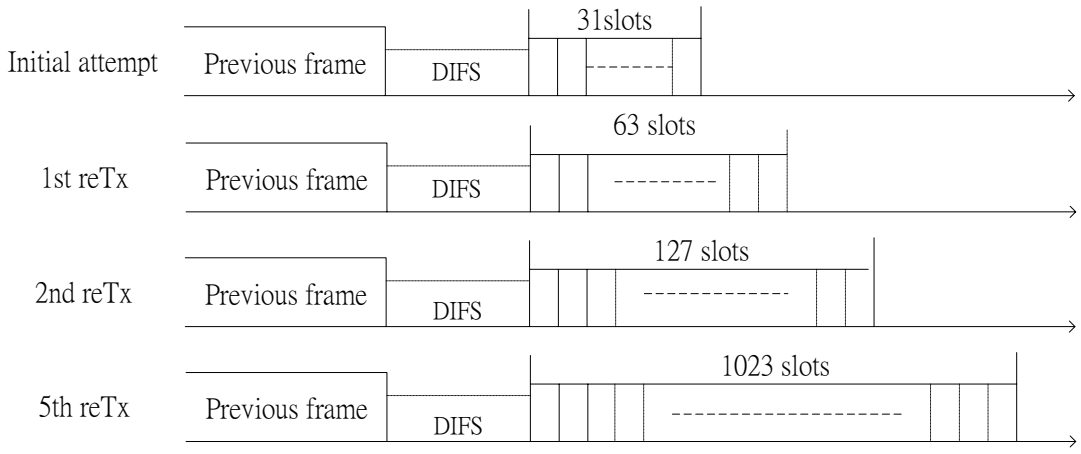


圖 5.28 Back-off window 示意圖

backoff window 會隨著嘗試使用 medium 失敗的次數(retry count)而增加，這是為了減少 Station collision 的機會.而當 backoff window 到了預設的上限就不會增加了，如同圖中第 5 次到第 6 次的情形.而當該 Station 搶到了 medium 存取權，backoff window 就會重置.如果該 Station 一直搶輸，到達一個預設的次數，那想傳送的這個封包就會當作被丟掉(discard).

5.4.1.2 Network Allocation Vector (NAV) & Request To Send/Clear To Send (RTS/CTS)

在成功地取得 medium 的使用權後，Station 發出的封包裡有個 NAV 的欄位，裡面的數值告訴其它人他會使用 medium 多久的時間，當然這段時間包括了數個 SIFS,傳 Frame 的時間以及該 Frame 的 ACK frame 時間等等的時間.其它人會從 NAV 開始倒數，只要還沒倒數至

0，都表示 medium 還在使用中.圖 5.29 是 NAV 功能的示意圖:

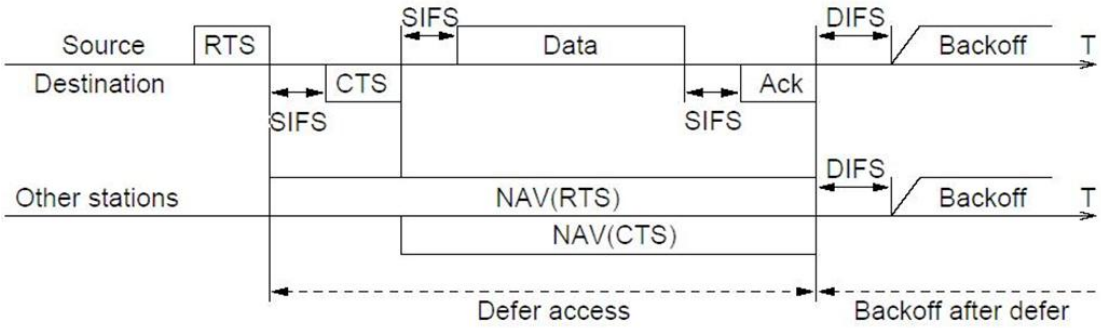


圖 5.29 NAV 示意圖

傳送端 Station 的 RTS 和接收端的 CTS 封包也有 NAV 的欄位.兩者的 NAV 長短不同，但結束的時間點一樣，會同時倒數至 0.聽到 RTS 和 CTS 的 Station 可以知道 medium 何時才能用，而不會在有人使的情況下去干擾別人.這個機制在多個網路重疊的情況下也有用，就算是屬於不同的網路，只要是用同一個 physical channel 的 Stations 都會看 NAV 來動作.

下圖是 RTS/CTS 運作的示意圖:

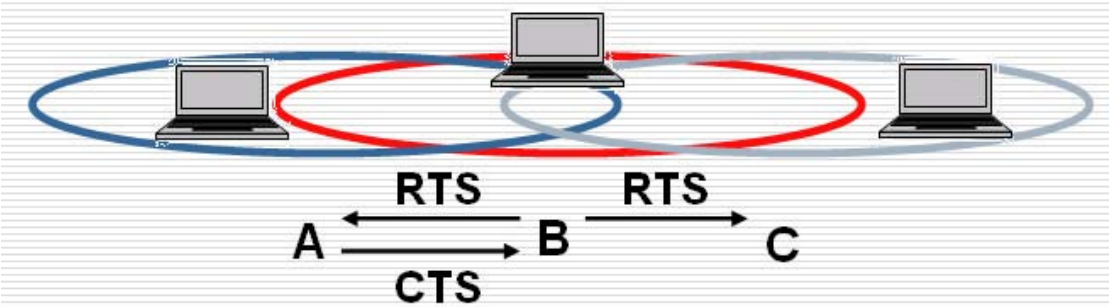


圖 5.30 RTS/CTS 運作示意圖

圖上兩個圓代表 Station A 和 Station C 訊號能送達的範圍.A

和 C 彼此無法知道對方的存在,但 Station B 在 Station A 和 Station C 的信號範圍內.

如果沒有 RTS/CTS， Station A 和 Station C 同時要和 Station B 通訊的話，會造成 collision. Station B 無法正常地和任一者通訊,但 Station A 和 Station C 也不知道問題出在哪; 但加了 RTS/CTS 的機制後，Station B 的 CTS 就會讓 Station C 知道 Station B 正在和別人傳輸資料中，Station C 就不會去干擾 Station B.

NAV 屬於 virtual carrier-sensing,是用來達到 Carrier sensing function 的一種方式， Carrier sensing function 的另一種方式是 physical carrier-sensing，此種方式會根據 medium 的不同和使用的調變方式不同而改變，在 RF-based media 的情況下， physical carrier-sensing 比 virtual carrier-sensing 貴得多， 所以多數的系統以 NAV 來作 Carrier sensing function.

如果傳送的封包長度大於 RTS threshold 這個參數，傳送端在傳送封包之前會先送出一個 RTS 的封包，RTS 會設定 NAV 長度到 Data 的 ACK 結束.接收端會回應一個 CTS，此 CTS 封包內的 NAV 也會設定到 Data 的 ACK 結束.RTS threshold 這個參數通常是在驅動程式的設定軟體中更改.

5.4.1.3 Fragmentation

802.11 為了減少同一個頻帶其它訊號的干擾，會對封包進行切割(Fragmentation).因為此頻帶的干擾源多是短時間，高能量地突然產生，所以把封包切成小段可以減少被干擾的封包的數量.至於多大的封包要被切割，取決於 fragmentation threshold 這個參數，大於這個長度的封包就會被切成數個小的封包再傳送.

NAV 的設定如下圖所示:

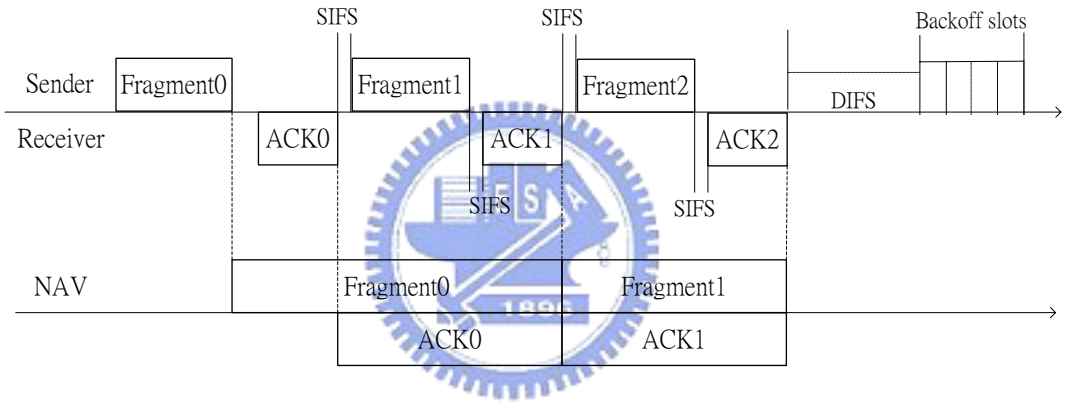


圖 5.31 封包經切割後再傳送之時序圖

只要不是最後一個小封包,都會把NAV設定到下一個小封包的ACK結束.而接收端的ACK封包也會設定NAV到下一個ACK結束;最後一個小封包會設定 NAV 到此封包的 ACK 結束，而接收端的 NAV 是 0.

RTS/CTS 再配合 fragmentation，的情形如下圖.

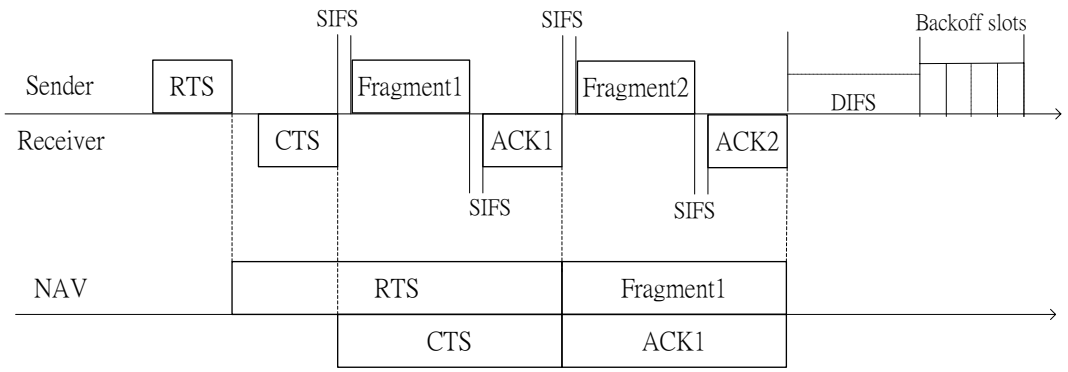


圖 5.32 RTS/CTS 搭配封包切割

比照沒有 RTS/CTS 的情況，可以看成 RTS 取代了傳送端的第一個小封包，而 CTS 取代了第一個小封包的 ACK.

5.4.1.4 Error Recovery

如果封包傳送中發生錯誤了，意即傳送端沒有收到接收端回傳的 ACK，那傳送端就要重送，每次重送會增加 retry count(沒有搶到 medium 使用權也會增加此 count). 當 count 到達一個預設的上限，就會丟棄這個封包，並向上層報告.Retry counter 又分為 short retry count 和 long retry count. Short 是指傳送的封包長度小於 RTS threshold; 反之就是 long.

Retry Count 重置只要滿足下列條件之一:

Short retry count:

1. 送出 RTS 後，收到對應的 CTS.

2. 送出 short 封包後，收到 ACK.

3. 收到 Broadcast or multicast 封包.

Long retry count:

1. 送出 Long 封包並收到 ACK.

2. 收到 Broadcast or multicast 封包.

會把 Retry count 分成兩種的原因是，網路管理者可以調低 Long retry count 來減少網路中較大的封包的數量，如此一來需要的 buffer space 也會比較小，在網路負荷比較重的時候，這麼做可以提高網路的強健度.

5.4.1.5 Point Coordination Function (PCF)

802.11 的標準也提供了一種叫 Point coordination function(PCF)的 medium 存取機制，這是一種為了提供即時服務而強制由 AP 來控制誰能有存取 medium 的權利.

在存取 medium 時有兩種模式:一種是 contention-free period，也就是 PCF 的機制;另一種是 contention period，是跑 DCF 的機制. 但現在市面上的產品很少有提供 PCF 的，所以我們在 PCF 的部份也不

再多著墨.

5.4.1.6 Broadcast and Multicast

雖然我們的傳輸平台並沒有使用廣播和群播的功能，但這裡還是提一下;在廣播和群播的情況下，傳送端不會需要接收端回 ACK，傳送的封包也不會被切割更小的 frame.因此廣播和群播都是單一步驟就能完成的傳輸動作，NAV 會設定成 0.如圖 4 所示: 當 medium 變為可用狀態持續了 DIFS 後，經過 backoff window 傳送端 Station 送出了一個廣播/群播的封包，NAV 是 0，所以傳完該封包後，經過一個 DIFS，其它 Station 就開始跑 contention window，準備下一次搶 medium 傳封包的程序.

5.4.2 無線環境下系統使用者數目上限之分析

在 5.3.4 小節，我們經由量測發現，當 client 數量超過 6 個之後，系統提供服務的情況變得不穩定，有時會顯示找不到 server，有時成功連上了 server，但實際上卻沒有傳輸資料.理論上 54Mbps 的 802.11g 網路頻寬應該要能容納 400 多個 130kbps 的連線,(Server 和一個 Client 間的串流會佔掉 130kbyte/sec 的頻寬，因為每 40ms 要傳出一張畫面.這張畫面會被分割成 5~6 個封包，一個封包是

1024byte)，但實測發現只能提供 6~8 個使用者同時使用，接下來我們試著分析其原因所在.

從上一個小節的 MAC 簡介中，我們知道傳送封包前會有 RTS/CTS，封包傳完之後還要回給傳送端一個 ACK.但在規格書裡並沒有講到 RTS/CTS/ACK 會花多少時間傳送，所以我們實際量了幾組數據，記錄 5 個連續的 1024byte 封包之間的間隔(包含了 RTS /CTS /DATA /ACK 全部所花的時間): 731us， 279us， 256us， 659us， 191us，這是從傳送端送到 AP 花的時間;若再加上 AP 傳到 client，又還要多一倍的時間，所以傳 5 個封包會用掉 2*(731+279+256+659+191)us = 4.23ms; 然而，無線網路的錯誤率比有線網路高許多(約每 20 個封包會錯 1 個)，導致重傳的情況很頻繁，所以傳送 5 個封包有可能會花掉 5ms 的時間.當然，這個數據只是一個例子，我們看了 10 組左右的數據，大概都會在 4~6ms 左右.一個 client 會用掉 4~6ms 的時間，所以每 40ms 最多只能處理 6~10 次，這也和我們觀測到的情況相符.

追根究柢，其實 802.11g 無線網路的頻寬並沒有 54Mbps 那麼高，以 5ms 傳了 5k bytes 的速度來計算，頻寬只有 8Mbps 左右.為什麼會減少這麼多呢?因為實際上傳送資料的頻寬需要扣掉 RTS /CTS /ACK /contention time 以及其它控制訊號的時間.且同一筆資料要先從 server 到 AP，再從 AP 到 client，等於同樣的資料用了兩次頻寬.

我們取 4 個觀測到的樣本,RTS/CTS/UDP data/ACK 花的時間如(表 5.3):

	RTS	CTS	Data	Ack	Data 所佔比例
樣本 1	166us	125 us	212 us	25 us	40.2%
樣本 2	385 us	131 us	204 us	9 us	28.0%
樣本 3	235 us	118 us	382 us	14 us	51.0%
樣本 4	491 us	71us	302 us	20 us	34.1%

表 5.3 data 傳送時間所佔百分比

除了 RTS/CTS/ACK 之外,還要考慮亂數決定的 contention time,也就是在 back-off window 花了幾個 slot time 當作 contention time,這也會降低實際會在傳 Data 的頻寬,最後可以得到 Data 所分到的頻寬 54Mbps * 1/2 * (28%~51%) = 7.56~13.5 Mbps,和之前我們得到的 8Mbps 相去不遠.

比較起在有線的網路環境,傳送一組封包(約 5~6 個)給一個 client 要花 1ms,40ms 約可分成 30~40 個 1ms,所以此系統可以支援到 30~40 個 clients.不過這是估測的數字,因為實際上沒有那麼多台電腦設備供實驗使用.

5.5 總結

在前面兩章已經看過 online 平順演算法的做法和效能，在這章我們希望這套做法可以實際增進傳輸系統的效能，重點是如何將演算法加入系統中.

5.1 小節介紹整個系統的架構，5.2 小節說明我們的程式是做在大

架構的哪一個區塊中，以及如何和系統互動.另外我們希望能夠找出系統可以容納的用戶數目上限以及網路對平順演算法的影響，所以做了許多觀測.在 5.3 小節我們將這些觀測結果做了整理並分析其原因.

5.4 小節更深入地探討無線網路的運作機制，在說明完 802.11 MAC 之後，我們試著分析系統用戶上限(無線網路環境下)比預期低很多的原因.

第六章 結論

本論文主要是延伸已有的 stored video 平順演算法，使其也能夠處理即時產生的影音資訊.我們一開始先回顧基本的平順演算法，說明其運作方式及特性,接著再探討即時資料和已存好的資料在平順處理上有何差異，針對這些不同之處改進原有的演算法.

從最單純的 fix size & non-overlapping window smoothing 到需要高運算量但效能更好的 sliding window smoothing，我們詳細地說明為何要做這些改變,最後可以發現,stored video 只是 online 把 window size 取成無窮大的一個特例.

為了最後能實作在傳輸系統上，原始的平順演算法必須修改成 packet-based 的規劃，才能相容於以封包為單位的傳輸系統.我們以模擬結果來說明演算法中各項參數對平順效能的影響,之後將此演算法實作到傳輸系統上時如何決定各項參數，就是跟據這些模擬結果.

在實作階段，需要對我們使用的傳輸平台有基本認識，有一小節是此傳輸系統的概要介紹.實作完成之後，在接收端觀測到的封包流量及到達時間和最初的規劃不完全相同,因此我們對系統所使用的無線網路做了關於延遲及 jitter 等的量測，並簡介 802.11 MAC，希望能解釋封包在經過網路前後多了哪些影響.

再來是未來可以努力的方向:目前系統只能處理影像，還需要加入聲音的部份才算完整.還有就是多個 client 同時存取同一台 camera 的影像時，最省力的作法是讓一個 thread 集中處理，再 multiplex 給各個 client;但目前的作法是多來一個 user 就多建一個 thread，這個部份還有改進的空間.

參考文獻

[1] J.D.Salehi ， Z.-L. Zhang ， J.F.Kurose ， and D. Towsley，” Supporting stored video: Reducing rate Varibility and end-to-end resource requirements through optimal smoothing，” IEEE/ACM Trans. Networking， vol. 6， pp. 397-410，1998

[2] S. Sen， J. L. Rexford， J. K. Dey， J. F. Kurose， and D. F. Towsley， “Online smoothing of variable-bit-rate streaming video，” IEEE Transactions on Multimedia 1997.

[3] W.Feng and S.Sechrest， “Smoothing and buffering for delivery of precorded compressed video，” in Proc. Of the IS&T/SPIE Symp. On Multimedia Comp. and Networking， pp. 234-242， Feb. 1995.

[4]Z.-L. Zhang， J. F. Kurose， J. D. Salehi， and D. Towsley， “Smoothing， Statistical Multiplexing， and Call Admission Control for Stored Video，” IEEE Journal on Selected Areas in Communications， Vol. 15， No. 6， August 1997， pp. 1148-1166.

[5] W.Feng and S.Sechrest “Critical bandwidth allocation for delivery of compressed video”， Comp. Comm.， vol. 18， pp. 709-717， Oct. 1995.

[6] O. Hadar， S. Greenberg， “Statistical Multiplexing and Admission Control Policy for Smoothing Video Streams Using e-PCRTT Algorithm，” The International Conference on Information Technology: Coding and Computing 2000， pp 272-277.

[7] W.-C. feng and J. L. Rexford， “Performance Evaluation of

Smoothing Algorithms for Transmitting Prerecorded Variable-Bit-Rate Video," IEEE Transactions on Multimedia, Vol. 1, No. 3, September 1999, pp 302-313.

[8] J. Zhang and J. Hui, "Applying Traffic Smoothing Techniques for Quality of Service Control in VBR Video Transmissions," Computer Communications, April 1998, pp. 375-389.

[9] S. Sahu, S.-L. Zhang, J, Kurose and D. Towsely, "On the Efficient Retrieval of VBR Video in a Multimedia Server," International Conference on Multimedia Computing and System 1997, pp. 46-53.

[10] J. M. Mcmanus and K. W. Ross, "Video-on-Demand Over ATM: Constant-Rate Transmission and Transport," IEEE Journal on Selected Areas in Communications, Vol. 14, No. 6, August 1996, pp. 1087-1098.

[11] W. Feng and J. Rexford, "A comparison of bandwidth smoothing techniques for the transmission of prerecorded compressed video," in Proc. IEEE INFOCOM, pp. 58-66, April 1997.

[12] W.-C. Feng, F. Jahanian and S. Sechrest, "An Optimal Bandwidth Allocation Strategy for the Delivery of Compressed Prerecorded Video," Multimedia Systems, Vol. 5, No. 5, 1997, pp. 297-309.

[13] Ming-Sheng Hsieh, "A smoothing Algorithm in Variable Bit Rate Streaming" A thesis for the Degree of Master of Science in

Communication Engineering， August 2004.

[14]W.-C. Feng， “Rate-Constrained Bandwidth Smoothing for the Delivery of Stored Video，” SPIE Multimedia Computing and Networking Conference Feb 1997.

[15]Chan-Wei Lin， “Smoothing Algorithm for Video Streaming in Packet Based Transmission System，” A thesis for the Degree of Master of Science in Communication Engineering， August 2005.