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

多執行緒在多媒體串流的應用

Construction of Multi-media Streaming
with Multithread Programing

研 究 生：韓孟潔

指導教授：張文鐘 教授

中 華 民 國 九 十 六 年 八 月

多執行緒在多媒體串流的應用

Construction of Multi-media Streaming with Multithread Programing

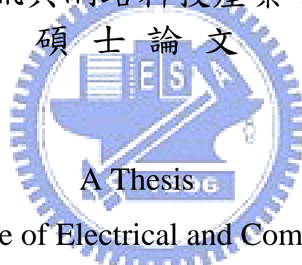
研 究 生：韓孟潔　　　　Student：Meng-Jie Han

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

多執行緒在多媒體串流的應用

研究生：韓孟潔　　　　　　　　　　　　　　　　　指導教授：張文鐘 博士

國立交通大學

電機學院通訊與網路科技產業研發碩士班

摘　　要

多媒體串流傳輸的技術,可以使得消費者不需花費大量的等待時間至檔案完全下載完畢即可播放影音檔案，此技術亦可使用在傳輸 live 的影音資料。

本論文探討利用 Java Media Framework 建置一個可以處理即時影音資料並傳送至遠方接收端的串流系統，系統建置流程如下：設置擷取裝置的路徑、設定影音來源的格式、影音分流、影音資料壓縮與 RTP 封包建立與發送。

此系統整體的運作是藉由多個 thread 各自執行專屬的任務,如截取影音資料、資料壓縮...等，以及多個 thread 彼此間相互溝通與合作而達成，在本論文中會探討 Java thread 同步機制的基本原理，以及實際系統程式所使用的 thread 排程機制，以達到 thread 之間的制衡。並實際觀測 multithread 的執行順序與時間，分析系統真實的狀態，並且說明 thread 之間確實是以 Java thread 同步機制和系統程式所使用的 thread 排程機制來運作，證明彼此之間互相牽制與合作達成系統所預期的功能。

Construction of Multi-media Streaming with Multithread Programing

Student：Meng-Jie Han　　　　　　　　Advisor：Wen-Thong Chang

National Chiao Tung University

Industrial Technology R & D Master Program on

Communication Engineering

ABSTRACT

With the multi-media streaming technology, consumers do not have to speed a lot of waiting time while downloading the multi-media data. This technology can also be used to transmit live audio/video.

This work discusses the construction of multi-media streaming system with Java Media Framework (JMF). System is constructed in the following sequence: set the path of capture device , set the file format of the audio/video, parse the multi-media data , compress multi-media data, create RTP Packets and transmit RTP Packets.

The Java based streaming system is operated with threads which may work alone or co-operate. Synchronization of Java threads and multithread scheduling are also discussed in this work. Finally, we observe the timing sequence of multithread.

誌　謝

能順利完成這篇論文，首先要誌謝的人是我的指導教授 張文鐘 博士。在這二年多的時間，老師在學術研究和研究態度、方法以及專業知識上，給予我許多的啟發，使我獲益良多，對於往後的發展幫助很大。同時也謝謝教授們於口試時的指導，有了您的指導才使得這篇論文更趨完備。

再來要感謝無線多媒體通訊實驗室的所有同學們，在這段期間內，大家一起討論、共同成長，謝謝你們的協助使我釐清許多盲點。跟你們在這兩年一起修課、聊聊生活瑣事、打球使得苦悶的研究生生活多了幾分亮麗、愉悅的色彩。

最後要感謝的是我的家人，你們不但感受我在這段時間內的壓力，更不時給予我最大的關懷與支持，有了你們的支持以及鼓勵我才能順利地完成碩士學業，在此向你們致上最高的感謝之意。

誌於2007.8 風城 交大

Meng Jie

目錄

中文摘要	……	i
英文摘要	……	ii
誌謝	……	iii
目錄	……	iv
圖目錄	……	vii
表目錄	……	x
一、	緒論……	1
1.1	背景介紹……	1
1.2	研究動機……	2
二、	Java 多媒體系統……	3
2.1	系統架構……	3
2.2	使用者介面(UI)之建構流程……	4
2.2.1	硬體支援之影音格式……	4
2.2.2	系統支援之影音格式……	5
2.2.3	影音編碼格式選項……	5
2.2.4	取得用戶端資訊……	8
2.3	子系統之建構流程……	8
2.3.1	影音來源……	8
2.3.2	影音分流……	9
2.3.3	影音壓縮……	11
2.3.4	RTP Session……	13
2.4	Java Media FrameWork……	15
2.4.1	JMF Architecture……	16

2.4.2 JMF RTP Architecture …… 20
2.5 Java Socket & Real-Time Transport Protocol …… 22
2.5.1 Java Socket …… 22
2.5.2 Real-Time Transport Protocol …… 25
2.5.2.1 RTP 檔頭架構 …… 26
三、 Java 多媒體系統動態分析 …… 28
3.1 各個 Thread 的功能介紹 …… 28
3.2 Thread建立的流程 …… 29
3.3 Thread之間的資料分享與溝通 …… 32
3.3.1 子系統介紹和 Thread 所屬之類別物件 …… 33
3.3.2 Thread的設計原理與制衡 …… 35
3.3.2.1 處理影像資料 …… 35
3.3.2.2 處理音訊資料 …… 42
3.4 系統程式執行的細部運作 …… 44
3.4.1 Video 資料的處理流程 …… 44
3.4.2 Audio 資料的處理流程 …… 49
3.4.3 資料壓縮與傳遞 …… 51
3.5 RTP封包建立與發送 …… 53
3.6 Java Thread …… 54
3.6.1 Thread狀態 …… 56
3.6.2 Thread 建立的方式 …… 57
3.6.3 Thread 同步機制 …… 58
3.6.4 Thread 之間的溝通 …… 60
3.6.5 自訂排程 …… 61
3.6.6 Multithread 之應用 …… 62

3.6.7 Java虛擬機的簡介 …… 63
四、 Thread試驗 …… 65
4.1 Thread封鎖與時間之試驗 …… 65
4.2 Thread執行次序和時間之量測 …… 68
4.2.1 暫存器之儲存列 …… 69
4.2.2 次序和時間量測 …… 69
4.3 結論與相關探討 …… 74
五、 結論 …… 78
參考文獻 …… 79
附錄一 Thread 順序和時間量測的數據 …… 80
附錄二 暫存器空間不足的特例 …… 81
附錄三 SourceThread(V)封鎖試驗 …… 83

圖目錄

圖 2-1 媒體資料處理系統 …… 3
圖 2-2 UI 建構流程 …… 4
圖 2-3 裝置支援格式 …… 5
圖 2-4 JMF 所支援的多工器 …… 6
圖 2-5 顯示 Video 編碼格式 …… 8
圖 2-6 處理媒體之子系統建構流程 …… 8
圖 2-7 JMF 支援的 parsers …… 10
圖 2-8 RTPSession …… 13
圖 2-9 JMF Architecture …… 16
圖 2-10 JMF Media Formats …… 17
圖 2-11 Process stages …… 18
圖 2-12 Processor states …… 19
圖 2-13 RTP transmission …… 21
圖 2-14 RTP reception …… 21
圖 2-15 JMF RTP architecture …… 21
圖 2-16 伺服端與用戶端之架構 …… 24
圖 2-17 Java Datagram Socket …… 24
圖 2-18 RTP architecture …… 25
圖 2-19 RTP data-packet header format …… 27
圖 3-1 Thread 建立流程 …… 28
圖 3-2 (a)音訊擷取裝置管理、音訊資料來源 …… 33
圖 3-2 (b)影像擷取裝置管理、影像資料來源 …… 33
圖 3-2 (c)影音來源之分流 …… 33
圖 3-2 (d)影音壓縮處理 …… 33

圖 3-2 (e)影音來源整合 …… 33

圖 3-2 (f) RTP 封包建立和傳送 …… 33

圖 3-3 (a)音訊來源 …… 34

圖 3-3 (b)影像來源 …… 34

圖 3-3 (c)影音分流 …… 34

圖 3-3 (d)多工器輸入來源 …… 34

圖 3-4 Thread Path(V) …… 36

圖 3-5 PushThread(V) …… 37

圖 3-6 VFWTransferDataThread(V) …… 38

圖 3-7 SourceThread(V) …… 39

圖 3-8 RawBufferStreamThread(V) …… 40

圖 3-9 buffer manage(V) …… 41

圖 3-10 Thread Path(A) …… 42

圖 3-11 DirectSoundPushThread(A) …… 43

圖 3-12 buffer manage(A) …… 44

圖 3-13 系統運作(V) …… 44

圖 3-14 擷取影像資料 …… 45

圖 3-15 抽取影像資料 …… 47

圖 3-16 資料壓縮流程 …… 48

圖 3-17 RTP 封包建立與發送之前置處理 …… 49

圖 3-18 系統運作(A) …… 50

圖 3-19 擷取&抽取音訊資料 …… 51

圖 3-20 資料壓縮與傳遞之流程 …… 52

圖 3-21 Video Packet Structure …… 53

圖 3-22 共用物件 …… 55

圖 3-23 Multithread…… 55
圖 3-24 Thread 物件的狀態圖…… 56
圖 3-25 thread 的集合 …… 59
圖 3-26 thread 間之溝通…… 62
圖 4-1 sleep(0)，CPU 之使用率 …… 66
圖 4-2 sleep(10)，CPU 之使用率 …… 67
圖 4-3 暫存器內容 …… 69
圖 4-4 順序和時間量測(a)(b)…… 70
圖 4-5 無檢查 EOM 時的執行順序 …… 75

表目錄

表格 2-1 擷取裝置資訊 …… 4

表格 2-2 DataSource 物件 …… 9

表格 2-3 BasicFilterModule 物件 …… 12

表格 2-4 Picture Formats Supported …… 12

表格 2-5 影音格式取樣率 …… 14

表格 2-6 Java Stream Socket API …… 23

表格 2-7 Java Datagram Socket API …… 24

表格 4-1 thread Sleep 試驗 …… 65

表格 4-2 暫存器內容 …… 72

表格 4-3 處理資料時間表 …… 73

第一章序論

1.1 背景介紹

由於近年來，網路科技和硬體的技術不斷的改進，網路服務變的更多元化，人們可以透過如PC、PDA、Smart Phone…等各式各樣的裝置，恣意地存取網路上任何地方的資源。透過網路已不只可作文字與音樂欣賞或是影像資訊的瀏覽，現而轉變成結合聲音、影像的互動式多媒體網路，這些多媒體資訊藉由多媒體串流技術的新傳輸科技在網際網路上迅速擴張。

過去，用戶端要下載一部短片，通常都需要等待一段時間，直到完全下載完畢後才能觀看。這會使用戶端花冗長的時間等待媒體資料的下載，是相當不方便的。多媒體串流技術則提供了解決此問題的方案。多媒體串流技術是具有即時性的傳輸技術，當用戶端要求下載媒體檔案時，會自伺服器端下載一小部分的媒體資料，即可立即播放，除了可避免用戶端等待冗長的下載時間、節省硬碟空間，同時可以提供傳輸live的影音資料和VOD(Video-on-Demand)的服務，也達到保護影音檔案的版權。

串流技術大概可分成三類，第一類以HTTP/TCP為基礎，利用HTTP協定可以讓串流媒體通過防火牆的阻礙，但因傳輸層所使用的為TCP通訊協定，因此，當傳輸的資料遺失時會要求重傳時，容易造成延遲(delay)。第二類是利用一獨立的串流伺服器，以RTP/UDP為傳輸的基礎，將多媒體資料送到使用者的播放器上播放。RTP提供即時性的端對端傳輸服務，包括payload type identification、sequence numbering…等，利用sequence number來進行封包重建，如此即可增進即時傳輸的品質。第三類串流技術叫Clientless Streaming，其是在串流過程中才將播放器送至用戶端，主要應用在行動裝置上，尤其是支援Java技術的平台。

目前多媒體的平台主流有 Apple 主推的 Quick Time Server、Microsoft 的 Media Services 以及 RealNetworks.com 的 RealSystem。在上述的三大陣營中，Apple 推出 QuickTime for Java 的 API，使得 Apple iPod 逐漸開啟了移動視頻

的市場。

近幾年來 Java 的版本不斷升級，並且釋出在各方面應用的套件，使得 Java 應用面很廣，如桌上型 PC 的應用程式、PDA 應用軟體的開發…等。Java 語言最大的好處就是在於可跨平臺、動態連結，及安全性的優勢，使其在網路電腦、家用電腦和嵌入式的控制器等領域中，成為最受歡迎的程式語言。

雖然 Java 的執行效能比 C/C++還要低了些，但近幾年 Java 加速器已逐漸改善 Java 的執行效能，以達到可以接受的水準。隨著網路的蓬勃發展，Java 作業系統和各種 Java 應用軟體所組成的 Java 網路資訊系統，在未來無線多媒體通訊的網路世界中將佔重要的地位。

1.2 研究動機

近幾年來，Java蓬勃發展，其跨平台的特性以及優異的物件導向語法，Java是非常適合用來發展系統、開發應用程式的語言。Java在多媒體串流上，有提供相關的套件，如Java Media Framework(JMF)為Sun與IBM合力制定的一個開放性與具可擴充性的多媒體處理API，其提供上層應用程式或Applet，是一種處理多媒體串流的一致性介面。JMF提供大部份標準的媒體編碼影音格式，但未包含MPEG4和H.264，但Apple推出QuickTime for Java的API 彌補JMF的不足，使得Java在多媒體方面的支援更加強大。

由於以上種種因素，我們對於 Java 在多媒體上的支援倍感興趣，Java 在多媒體方面釋出以下幾種套件:Java Media Framework、Java 3D、Java 2D…等。其中 Java Media Framework(JMF)套件可以做得到的功能相當的多，如擷取影音檔案、http 檔案下載和播放，也可以做到傳輸串流媒體，JMF 算是相當完整的多媒體套件。藉此本論文將會探討 JMF 對處理多媒體資料和即時傳輸所提供的套件，以及程式如何實際建立與執行。

第二章 Java 多媒體系統

2.1 系統架構

本論文利用 JMF 所提供的 API 套件(請參考 2.4.1 節),探討一個 Java 多媒體資料處理系統,此系統主要目的為傳送擷取的即時性影音媒體資料至遠方用戶端,並且利用 JMF 所提供的 RTP/RTCP 的套件(請參考 2.4.2 節),以串流的傳輸形式將資料送至用戶端。

整體系統的運作是藉由 9 個 thread 在處理,其中包括擷取裝置管理、媒體資料來源、影音來源之分流、影音壓縮處理、影音整合與 RTP 封包建立和傳送,都會有專門的 thread 處理,在第三章會詳細說明。

以下為傳輸端的系統架構:

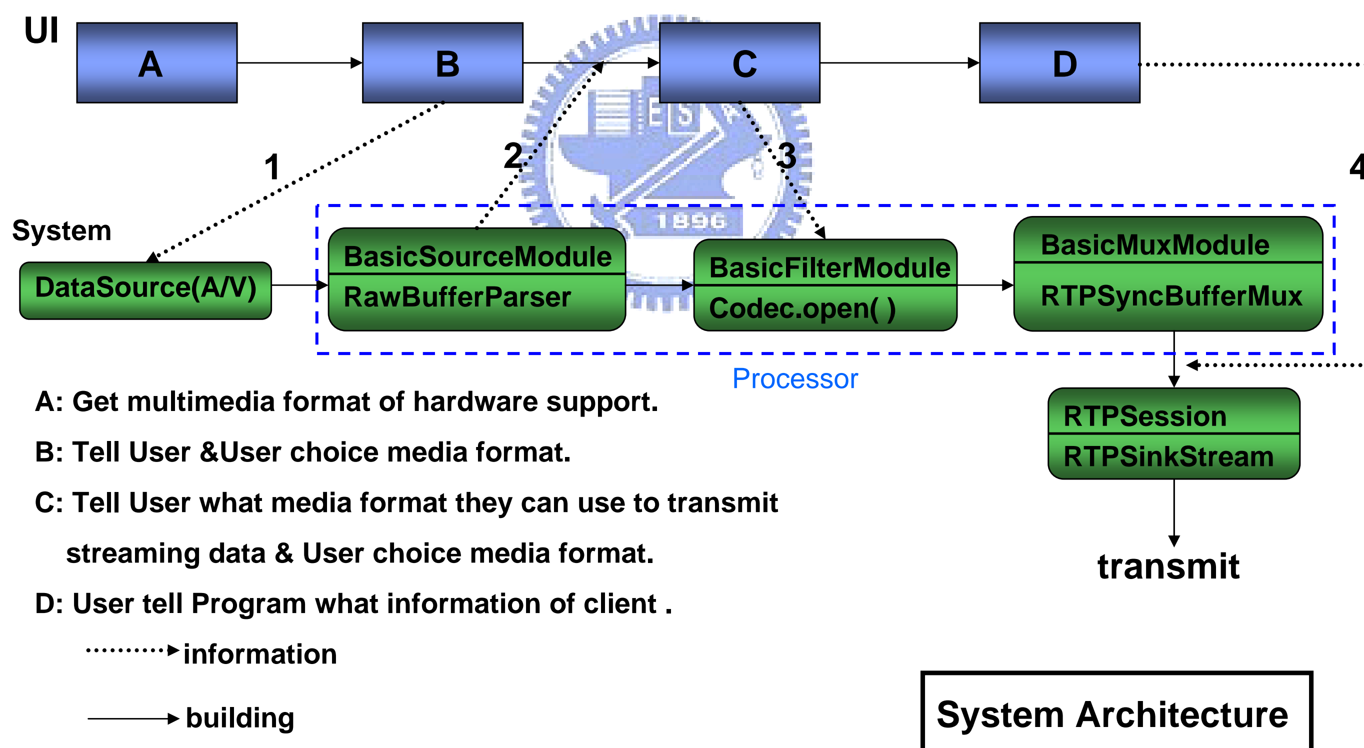


圖 2-1 媒體資料處理系統

圖 2-1 中,藍色方塊是描述使用者介面(UI)的建構流程,其表示傳送端的系統程式與本地端使用者之間的互動;建構子系統的同時必須向使用者取得建構子系統所需的訊息。綠色的子系統方塊主要為處理媒體資料和傳送串流媒體至遠方

用戶端。圖 2-1 中，information:代表建立子系統時，所必須自 UI 取得的相關資訊；同時建立 UI 時，所需自子系統取得的相關資訊。building: UI 和子系統方塊建立的流程。

2.2 使用者介面(UI)之建構流程

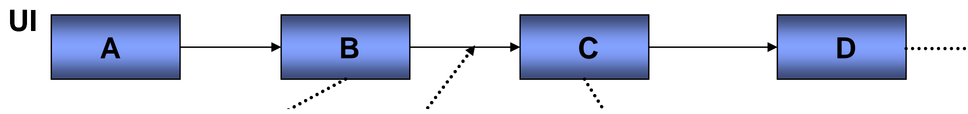


圖 2-2 UI 建構流程

2.2.1 硬體支援之影音格式

圖 2-2，A 之方塊中，其主要為取得關於擷取裝置的資訊，相關資訊包括擷取裝置的位置、支援哪些影音格式。JMF 套件有提供 query mechanism(請參考 2.4.1)，以得知擷取裝置的資訊，因此必須執行 CaptureDeviceManager. getDeviceList( )函式，取得包含擷取裝置的相關資訊之物件:CaptureDeviceInfo (包含 VFWDeviceQuery)物件，CaptureDeviceInfo 物件內容主要為:裝置名稱、裝置位置、此裝置所支援的格式、音訊的取樣率與影像的大小。

表格 2-1 舉出 CaptureDeviceInfo 的部分內容：

表格 2-1 擷取裝置資訊

音訊之相關資訊(CaptureDeviceInfo)	
裝置名稱 (MediaLocator)	DirectSoundCapture : dsound://
音訊格式/ 取樣率	LINEAR, 48000.0 Hz, 16-bit, Stereo, LittleEndian, Signed
影像之相關資訊(VFWDeviceQuery)	
裝置名稱 (MediaLocator)	vfw:Microsoft WDM Image Capture (Win32):0 : vfw://0

影像格式 /影像大小	YUV Video Format: Size =java.awt.Dimension[width=160,height=120], MaxDataLength = 28800
	RGB, 160x120, Length=57600, 24-bit

當此階段無法取得CaptureDeviceInfo物件時，判定無可以使用的擷取裝置時,會顯示"No capture devices found in JMF registry!"的對話視窗。

2.2.2 系統支援之影音格式

圖 2-2，B 之方塊中，其主要為建立一對話視窗，將系統有支援的影音格式告知使用者(本地端)，使用者可依其需求選擇影音格式、音訊取樣率與影像大小。對話視窗顯示兩資訊:音訊和影像資訊。在音訊方面，必須呼叫 CaptureDeviceInfo.getFormat()函式，取得一存有音訊格式型態的陣列，此陣列包含擷取裝置所支援的輸出音訊格式目錄。將此目錄裡的資訊一一抽出，做為建立音訊的訊息對話視窗，告知使用者可選用的音訊格式。

在影像方面，必須呼叫 VFWDeviceQuery.getFormat()函式，取得一存有影像格式型態的陣列，此陣列包含擷取裝置所支援的輸出影像格式目錄。同樣將此目錄裡的資訊一一抽出，做為建立影像的訊息對話視窗，告知使用者可選用的影像格式。如下圖 2-3:

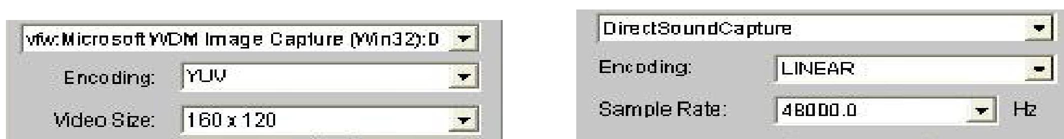


圖 2-3 裝置支援格式

當使用者完成選定影音格式的動作後，其相關資訊(圖 2-1 的虛線 1)會用來做 DataSource 物件的初始與建立(2.3.1 節會說明此物件的功能)。

2.2.3 影音編碼格式選項

由圖 2-1 中，經由使用者選定擷取裝置輸出的媒體格式，建立 DataSource

物件，接著會選用適當的 parser，同時建立 processor(以上流程在 2.3.2 節說明)。當 parser 選定後，會依 parser 輸出的內容格式(content type)找出相對應的多工器，此多工器的輸入影音格式會成為建立 C 之方塊時所需的資訊(圖 2-1 的虛線 2)。圖 2-2，C 之方塊中，主要為告知使用者，影音編碼的傳輸格式(也就是多工器的輸入影音格式)有哪一些，由使用者選定後，影音資料將來會壓縮為使用者所選定的影音編碼格式(圖 2-1 的虛線 3)。

以下為取得影音編碼格式選項的程序:

(1)找出 JMF 支援的多工器

執行 pluginManager.getPlugInList (,,5)函式(請參考 2.4.1)，取得所有確定存在的多工器，總共有 10 個。如下圖 2-4:

Name	Value
elementCount	10
elementData	Object[10] (id=1312)
[0]	ClassNameInfo (id=1279)
className	"com.sun.media.multiplexer.RawBufferMux"
hashValue	-2480469308525409387
[1]	ClassNameInfo (id=1313)
className	"com.sun.media.multiplexer.RawSyncBufferMux"
hashValue	-1443959470845784203
[2]	ClassNameInfo (id=1314)
className	"com.sun.media.multiplexer.audio.GSMMux"
hashValue	282133703800236816
[3]	ClassNameInfo (id=1315)
className	"com.sun.media.multiplexer.audio.MPEGMux"
hashValue	2471369822517220134
[4]	ClassNameInfo (id=1316)
className	"com.sun.media.multiplexer.audio.WAVMux"
hashValue	282133704876461107
[5]	ClassNameInfo (id=1317)
className	"com.sun.media.multiplexer.audio.AIFFMux"
hashValue	2471369791244919035
[8]	ClassNameInfo (id=1320)
className	"com.sun.media.multiplexer.video.QuicktimeMux"
hashValue	7877271409392625748
[9]	ClassNameInfo (id=1321)
className	"com.sun.media.multiplexer.RTPSyncBufferMux"
hashValue	-2095741743343195187
modCount	10

圖 2-4 JMF 所支援的多工器

(2)建立 ContentDescriptor 物件

執行 processor.getSupportedContentDescriptors ()函式(請參考 2.4.1 節)，此函式會取得媒體資料格式。前面有提到，在此主要處理擷取裝置的影音資料，因此此函式所取得的影音資料的內容型別(content type)為 raw，此時意味著所取得的影音資料為未經過壓縮處理的媒體資料。利用前面所取的內容型別加上".rtp"，接著建立 ContentDescriptor 物件，其包含內容型別為"raw.rtp"，意謂著經處理後的媒體資訊是要送至網際網路。

(3)找尋適當的多工器

1.執行 processor.setContentDescriptor(contentDescriptor)函式，設定此時 processor 的輸出內容型別為"raw.rtp"，此函式會詢問 PlugInManager，因此會呼叫 PlugInManager.getPlugInList(,contentDescriptor, 5)函式，詢問在所有多工器中是否有內容型別為"raw.rtp"的多工器。以下為找尋適當多工器的過程：

將"raw.rtp"轉為某一整數。而所有的多工器會將它的輸出格式(含有各自的內容型別)轉為整數，接著依序與"raw.rtp"所對應的整數做比對，直到找到多工器的內容型別與此整數互相符合，在此所找到的多工器是 com.sun.media.Multiplexer.RTPSyncBufferMux，此多工器是負責處理要送至網際網路上的媒體資料。

2.執行 RTPSyncBufferMux.setInputFormat(input[],TrackID)函式，input[ ]為一陣列，其存放 processor 的輸出媒體格式:codec 的列表；TrackID 為音訊或影像的 TrackID。此函式會將 RTPSyncBufferMux 所能提供的媒體輸入格式與 RTP Payload Type 作比對，互相符合的格式資訊會存入陣列中，接著利用所得到的資訊建立影音編碼格式選項的對話視窗，如圖 2-5。

影音編碼格式選項的對話視窗，其告知傳輸端使用者下面資訊:

影像編碼格式選項:h263/rtp 和 jpeg/rtp。

音訊編碼格式選項: dvi/rtp、mpegaudio/rtp、ulaw/rtp、gsm/rtp、g723/rtp。

圖 2-5　顯示 Video 編碼格式

2.2.4 取得用戶端資訊

圖 2-2，D 之方塊中，其主要為建立一取得用戶端資訊的對話視窗。對話視窗要求傳送端的使用者，輸入使用者所要傳送的目的地之 IP 位址、port，當使用者輸入資訊後，會執行使用者所輸入接收端的 IP Address、Port 是否正確的檢查程序，若有錯誤則會顯示通知使用者"有錯誤的"的對話視窗，並且對話視窗不會再往下一頁跳，仍是在原先的視窗等待使用者重新輸入資訊。圖 2-1 中，我們可知，使用者所輸入的資訊是將來要建立 RTPSession 所需的資訊((圖 2-1 的虛線 4))。

2.3 子系統之建構流程

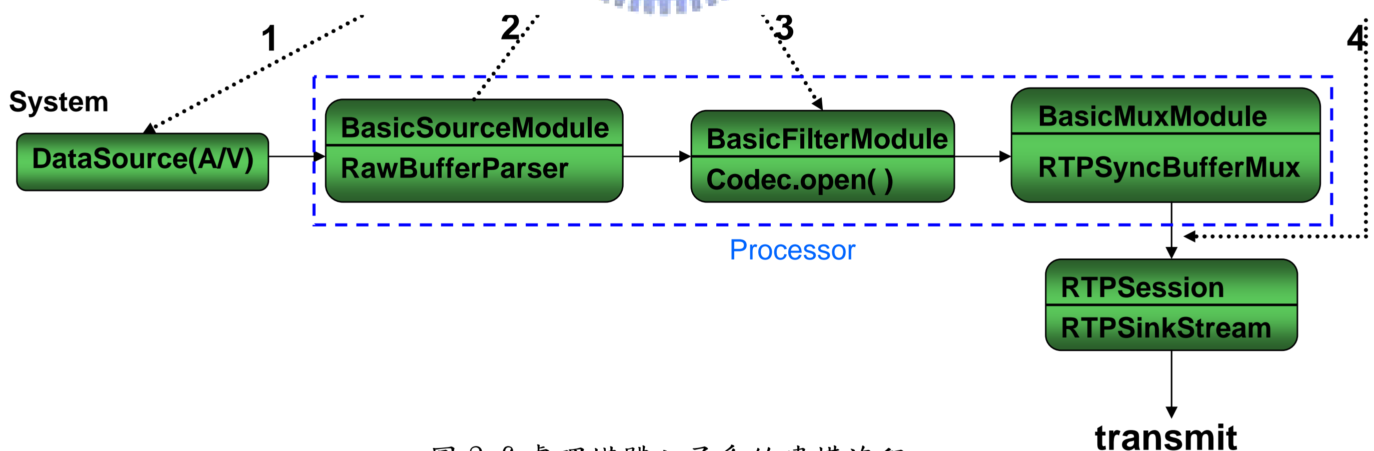


圖 2-6 處理媒體之子系統建構流程

2.3.1 影音來源

藉由 UI 所傳遞的資訊(圖 2-6 的虛線 1):影音資料格式和影音來源位置，建立管理影音資料的 DataSource 物件，如圖 2-6，必須執行下面函式(請參考 2.4.1):

JMFUtils.createCaptureDataSource(audioDeviceName, getAudioFormat( ), videoDeviceName, getVideoFormat( )),其函式利用已知的裝置名稱和影音媒體格式,建立 DataSource 物件,如表格 2-2。

表格 2-2 DataSource 物件

DataSource
streams:PushBufferStream[2]
Connect()

DataSource 物件內的 PushBufferStream 陣列,會有存有管理影像資料和音訊資料的物件,分別是 VFWSourceStream 和 DirectSoundStream 物件。connect( )函式與 thread 的建立有關,請參考 3.2 節。

2.3.2 影音分流

有了影音資料的來源,我們必須利用 DataSource 物件作為引數,建立圖 2-6 中的 Processor,接著必須執行 Manager.createProcessor(DataSource)函式。媒體資料將會輸入 Processor,經過各個處理媒體資料的元件,執行相關資料處理的程序。當媒體資料傳遞至 Processor,則必須將影音資料執行影音分流的處理,因此必須執行以下程序,(1)找出適當的 parser(解多工器),執行影音分流的處理 (2)建立 BasicSourceModule 物件,此物件作為媒體資料壓縮處理的來源。

(1)找出適當的 parser。DataSource 物件包含媒體的內容型別,由於影音來源自擷取裝置,因此內容型別為 raw,藉由此內容型別找出相對應的 parser。首先建立 ContentDescriptor 物件,其包函媒體的內容型別 (raw),接著執行 PlugInManager.getPlugInList(ContentDescriptor, ,1)函式,此函式會找尋所有 JMF 有支援的 parsers(圖 2-7),最後將內容型別 (raw)和所有 parsers 的輸入的格式都轉為數值,接著開始互相比對,互相符合者就是我們所要使用的 parser,

在此為 RawBufferParser，接著建構 RawBufferParser 物件，此 parser 是負責處理未經過處理的媒體資料。

Name	Value
elementData	Object[20] (id=1024)
[0]	ClassNameInfo (id=1036)
className	"com.ibm.media.parser.video.MpegParser"
hashValue	7411937007509527934
[1]	ClassNameInfo (id=1025)
className	"com.sun.media.parser.audio.WavParser"
hashValue	6780268352703333930
[2]	ClassNameInfo (id=1027)
className	"com.sun.media.parser.audio.AuParser"
hashValue	7144025166542677540
[3]	ClassNameInfo (id=1028)
className	"com.sun.media.parser.audio.AiffParser"
hashValue	747112652926045086
[4]	ClassNameInfo (id=1029)
className	"com.sun.media.parser.audio.GsmParser"
hashValue	6780213838714321907
[5]	ClassNameInfo (id=1030)
className	"com.sun.media.parser.RawStreamParser"
hashValue	-5775026903496749038
[6]	ClassNameInfo (id=1031)
className	"com.sun.media.parser.RawBufferParser"
hashValue	-8796373525217104636
[7]	ClassNameInfo (id=1032)
className	"com.sun.media.parser.RawPullStreamParser"
hashValue	-5143466831779007343
[8]	ClassNameInfo (id=1033)
className	"com.sun.media.parser.RawPullBufferParser"
hashValue	-8164813453499362941
[9]	ClassNameInfo (id=1034)
className	"com.sun.media.parser.video.QuicktimeParser"
hashValue	5091550473743594668
[10]	ClassNameInfo (id=1035)
className	"com.sun.media.parser.video.AviParser"
hashValue	5963159313270237981

圖 2-7 JMF 支援的 parsers

(2)建立 BasicSourceModule 物件。利用上述的 parser 和 DataSource 物件以引數的形式，呼叫 BasicSourceModule(DataSource, parser)，建立 BasicSourceModule 物件，此物件負責將音訊或影像資料送至處理壓縮媒體資料的子系統，請參考 3.4 節。

當 Processor 已得到關於媒體資料的資訊，確定資料的格式與位置，

Processor 的狀態應該轉變為 Configured 的狀態(請參考 2.4.1 節)。因此必須呼叫 configureProcessor ()函式，主要執行下面程序:執行 parser 的 getTracks() 函式，會依 track 的數量建立相同數量的 RawBufferParser$FrameTrack 物件: BufferTransferHandler，此物件將會分別抽出音訊和影像的媒體資料，負責將媒體資料送入 parser。

最後必須要對 DataSource 物件設置負責資料傳遞的 BufferTransferHandler，分別下面函式:
VFWSourceStream.setTransferHandler(RawBufferParser$FrameTrack)、
DirectSoundStream.setTransferHandler(RawBufferParser$FrameTrack)，
BufferTransferHandler 是負責將媒體資料自 DataSource 物件抓入 Parser 內的暫存器，實際資料的搬運細節請參考 3.4。

2.3.3 影音壓縮

欲將媒體資料以串流的傳輸方式傳送至用戶端前，必須將未處理的媒體資料壓縮，轉換為能進行即時串流傳輸的媒體格式。由 2.2.3 節，自 UI 取得影音編碼格式的資訊，由此資訊選擇所要使用的 Codec，Codec 以引數的形式建立 BasicFilterModule 物件，此物件是負責將未經處理的媒體資料交由 encoder 處理，編碼完成後的媒體資料會送入多工器，由多工器將媒體資料送至 RTPSession。

因此必須將 Processor 狀態由 Configured 更新為 Realized(請參考 2.4.1)，更新 Processor 狀態必須執行 realizeProcessor()函式，此函式會建立 RealizeWorkThread 物件，此 thread 進入待命狀態，當開始執行時會呼叫 doRealize()函式，來執行以下兩個程序:1.建立 BasicFilterModule 物件，2.建立多工器。

1.Codec 以引數的形式建立 BasicFilterModule 物件，如表格 2-3。

表格 2-3 BasicFilterModule 物件

BasicFilterModule
codec:Codec ic:BasicInputConnector oc:BasicOutputConnector

表格 2-3 中 ic 為媒體資料輸入 BasicFilterModule 的路徑，oc 為自 BasicFilterModule 輸出的路徑，ic 和 oc 是輸入/出 BasicFilterModul 的連結器，它的內部會定義 buffer 和媒體輸入/出的格式。接著呼叫 BasicFilterModule. doRealize()函式，此函式會執行 codec.open()函式，對 encoder 作初始。

表格 2-4 Picture Formats Supported

FRAME FORMAT	Width	height	nativeformat
SQCIF	128	96	0
QCIF (PAL)	176	144	1
CIF (PAL)	352	288	2

在此以 H.263 為例，表格 2-4 中，提供 3 種影像格式，藉由影像資料欲壓縮的影像格式(大小)決定 nativeformat 參數值，初始 h.263 的 encoder。由 2.2.2 節，自 UI 取得擷取的影像大小的資訊，接著對影像大小的資訊作一簡單的檢驗程序，如下:

```
if   videosizewidth>=352
{   w=352;
    h=288;
} else if videosizewidth>=160
{   w=174;
    h=144;
}else
{   w=128;
    h=96;    }
```

初始 h263 的 encoder 所需的參數有:**nativeformat**、**MTUPacketSize** 與 **IFramePeriod**。**MTUPacketSize** = 1024-**RTPsize**(12)-**UDPsize**(8)-**IPsize**(20) = 984(**byte**)，由於編碼完成的資料將來都必須封裝程 **RTP** 封包，而在 **Internet** 所傳遞的封包大小不可以超過 1500**byte**，因此每一編碼完成的資料都會存入暫存器中，暫存器所存的資料就是 **payload**。每一 **Picture layer** 內總共有 15 個 **frame**，第一個 **frame** 為 **Iframe**，因此 **IframePeriod** 設為 15。

2.執行 **connectMux**( )函式，此函式建立 **BasicMuxModule** 物件，此物件會包含 **RTPSyncBufferMux**(多工器)物件，接著初始多工器的輸入端，因此建立兩個 **RawBufferSourceStream** 物件，此物件是媒體(**A/V**)輸入多工器，資料存放的地方。當建立 **RawBufferSourceStream** 物件的同時，會建立兩個 **RawBufferStreamThread**，此兩個 **thread** 負責傳遞實際的音訊和影像的資料，請參考第三章。

2. 3. 4 RTP Session

此節，將會探討如何建立**RTPSession**以及其流程，如圖2-8。

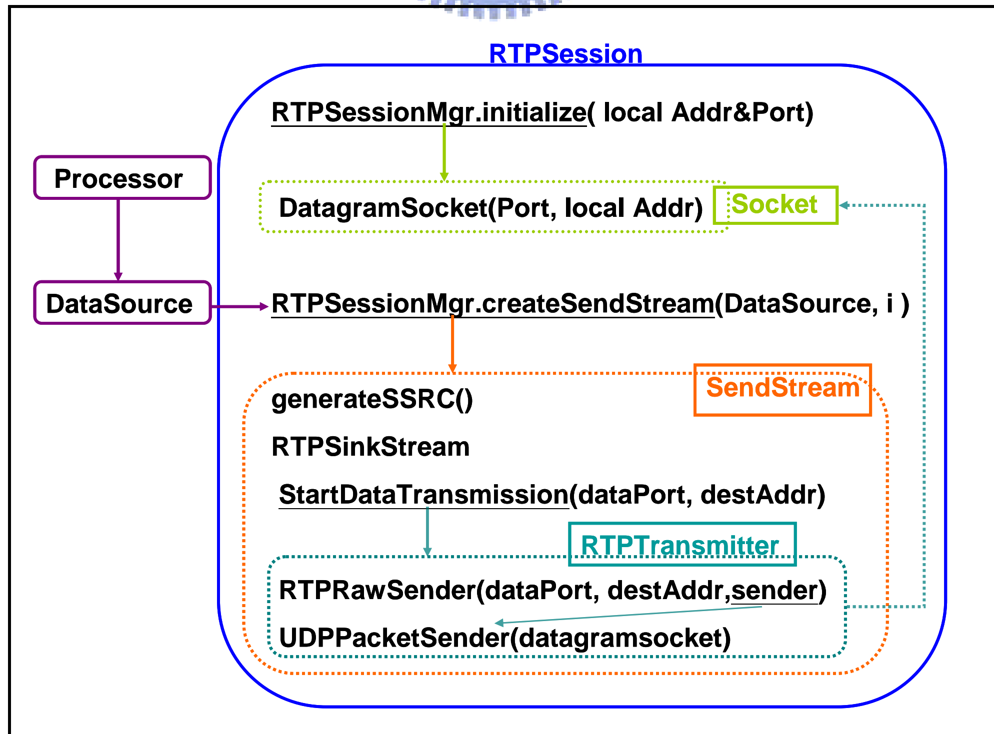


圖2-8 **RTPSession**

即時的影音資料經Processor(圖2-1)處理後，以串流的傳輸方式將媒體資料傳送至遠方的用戶端，為達到此目的我們必須建立RTPSession，來管理傳送RTP封包的session(請參考2.4.2節)。

利用源自UI所取得遠方用戶端的資訊:IP位址和port來建立RTPSession，必須執行JMFUtils.createSessionManager (IP位址, Port, Ttl, null)函式，此函式會負責建立RTPSessionMgr物件，由此物件管理RTPSession，如圖2-7。當欲自本地端傳送封包至網際網路時，必須先建立Socket，由2.5.1節可知，負責傳送封包的為DatagramSocket，因此必須給予專屬此DatagramSocket的port和IP位址，必須執行RTPSessionMgr.initialize(本地端IP位址,port)，此函式會建立DatagramSocket，DatagramSocket會被繫結至本地通訊埠，發送封包時，該通訊埠也會被填入datagram的標頭。

由圖 2-7 可知，自 Processor 輸出的媒體資料是包含在 DataSource 物件，其中 i 引數是代表要對音訊或是影像建立資料串流。接著要對同一音訊或影像串流設定一 SSRC 的數值，執行 generateSSRC( )函式。由圖 2-1 我們已知 Processor 的輸出也就是多工器的輸出，當媒體資料自多工器輸出時必須要有一暫存器存放輸出的資料，RTPSinkStream 物件就是扮演上述的角色，因此建立 RTPSinkStream 物件。並且執行 RTPSinkStream.setFormat(format)，format 物件為 RTPPayload 的影音格式，接著呼叫 format.getSampleRate( )函式取得此資料型態的取樣率。

舉例來說若 format 為 H.263 的格式則得到的取樣率就為 90000HZ，表格 2-5 列舉在 JMF 有支援的影音格式相對於取樣率的列表:

表格 2-5 影音格式取樣率

Payload type	JMF 有支援的編碼標準	支援 Audio 或 Video	Clock Rate(Hz)
0	ULAW/RTP	A	8000
3	GSM/RTP	A	8000
4	G723/RTP	A	8000
5	DVI/RTP	A	8000

14	MPEGAUDIO/RTP	A	32000
15	G728/RTP	A	8000
26	JPEG/RTP	V	90000
31	H261/RTP	V	90000
32	MPEG/RTP	V	90000
34	H263/RTP	V	90000

當 RTPSinkStream 物件取得媒體資料後，必須將媒體資料封裝為 RTP 封包，設置 RTP 標頭所有參數(請參考 2.5.2)，RTPTransmitter 物件會執行上述功能。RTP 封包建立完成後，將資料填入 DatagramPacket，接著 DatagramPacket 會由 DatagramSocket 送出，如圖 2-7 中，RTPRawSender(dataPort, destAddr,sender)所建立的物件會執行上述流程，其中 dataPort:通訊埠，destAddr:用戶端 IP 位址，sender 為 UDPPacketSender(datagramsocket)，UDPPacketSender 物件負責將 DatagramePacket 送至 DatagramSocket。

最後執行 SendStream.start( )函式，當呼叫 start( )函式時則會執行 RTPSinkStream.start()此時表示已準備好可以將影音資料自 Processor 送入。實際媒體資料傳遞細節，請參考 3.4 節。

2.4 Java Media FrameWork

Java Media FrameWork (JMF)提供一組能夠處理 time-base media data 的多媒體套件，它提供了擷取、處理與傳送的統一建構方式。而 time-base 提供的資訊只有目前的時間，以 system clock 為主。JMF 能提供大部份標準的媒體內容型別(content type)，例如:AIFF、AU、AVI、MPEG、WAV、QuickTime 和 WAV 等。

JMF 主要為兩大部份，第一部份為處理多媒體資料的相關套件，第二部份為 Real-Time Transport protocol 相關的套件。以下會分別說明與介紹。

2.4.1 JMF Architecture

JMF 提供兩種型態的 API(圖 2-9)，一為高階 API: 提供擷取、處理與呈現媒體資料的功能。另一種為低階 API:提供設計者擴充元件的功能。

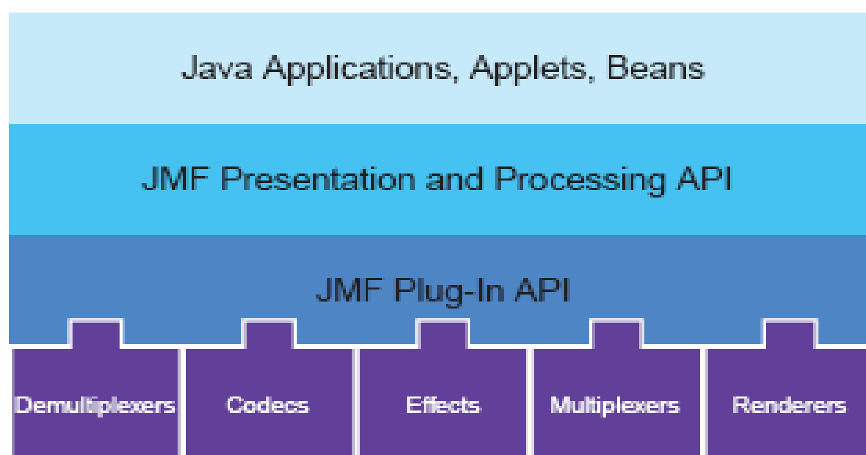


圖 2-9 JMF Architecture

JMF 處理媒體的介面大致分為以下幾種:

- 時間模型(Time Model):串流媒體的媒體時間，其可表示為媒體播放的位置。
- 管理者(Managers)，主要有以下 4 種類別，其都置於 javax.media 套件內，主要功能如下:
  - Manager:負責建立 Processor、DataSource。
  - PackageManager:管理已註冊的套件,這些套件包含 JMF class,如 Players、Processors...等，這些元件也可為自行設計。存放 package-prefix 列表,將來 manager 會依照列表找出 time-based media 的協議處理者和內容處理者.
  - CaptureDeviceManager:使用一 registry 和 query 機制，找出擷取裝置的位置，以及對於可使用的裝置回傳 CaptureDeviceInfo 物件。CaptureDeviceManager(位置於 javax.media 套件中)通常被使用來註冊新的擷取裝置.。
  - PlugInManager:管理 JMF 所有可以使用的 plug-in processing 元件，如

Multiplexers、Demultiplexers、 Codecs、Effects、Renderers。

➢ 資料模型(Data Model)

JMF 通常是使用 DataSource 來管理媒體內容的轉換、連結，DataSource 內會包含媒體的來源位置、媒體資料的格式與傳輸協定等。JMF 將 DataSource 主要分為二種類型：

- Push 和 Pull 資料來源:

  此類型的 DataSource 可根據資料傳送方式分為二類:

  ✓ PullDataSource：接收端開始傳遞資料，並控制 pull data-sources 的資料流，使用的 protocols 型態為 HTTP、FILE。JMF 定義二種 pull data sources 型態：PullDataSource、PullBufferDataSouce。

  ✓ PushDataSource：傳送端開始傳遞資料，並控制 push data-sources 的資料流。使用的 protocols 型態包括 RTP、multicast media、VOD。JMF 定義二種 push data sources 型態：PushDataSource、PushBufferDataSouce

  一個標準的 data source 是使用 byte 為傳送單位；一個 buffer data source 是使用 buffer 為傳送單位。

- Data Format: JMF 描述音訊和影像格式的屬性，音訊格式如取樣率,channel 的數量。影像格式如下圖 2-10。

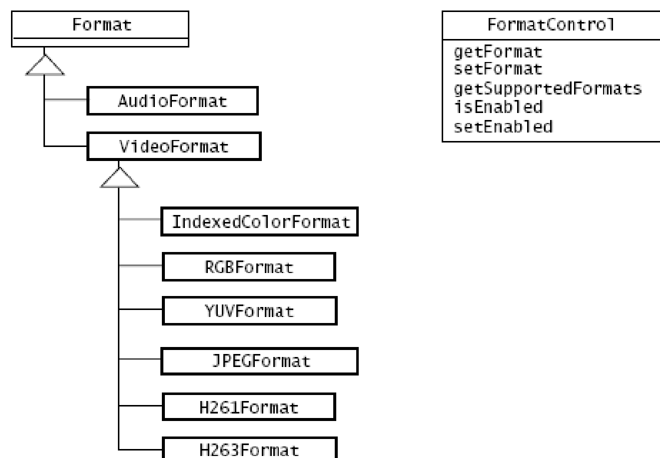


圖 2-10 JMF Media Formats

JMF 提供許多處理媒體資料的套件，以下依 JMF 所提供的功能，分別加以說明。

➢ Capturing

擷取裝置可以扮演媒體資料的來源，例如麥克風和攝影機。這些擷取裝置可作為抽象的 DataSource，如擷取裝置可傳遞具有時間性的媒體資料，此 DataSource 為 PushDataSource。擷取裝置可傳遞複合性資料串流(multiple data streams),其 DataSource 是複合性 SourceStreams,它會對應到裝置所提供的資料串流。

要取得擷取裝置的相關資訊，必須透過 CaptureDeviceManager,存取關於擷取裝置的資訊，執行 CaptureDeviceManager.getDeviceList( )函式可以得到所有可利用的擷取裝置列表。CaptureDeviceInfo 物件描述每一個裝置,取得 CaptureDeviceInfo 物件,如下例:

```
CaptureDeviceInfo deviceInfo = CaptureDeviceManager.getDevice("deviceName");
```

要從裝置上擷取多媒體資料，必須由它的 CaptureDeviceInfo 物件取得裝置的 MediaLocator 物件。

➢ Processing

Processors 可以用來播放媒體資料，除此之外，Processor 也可以使得自 DataSource 所輸出的媒體資料，由其它的 Player 和 Processor 來呈現，或是將媒體資料傳送到其他的目的地，如網際網路。媒體資料處理分為幾個階段，如下圖 2-11。

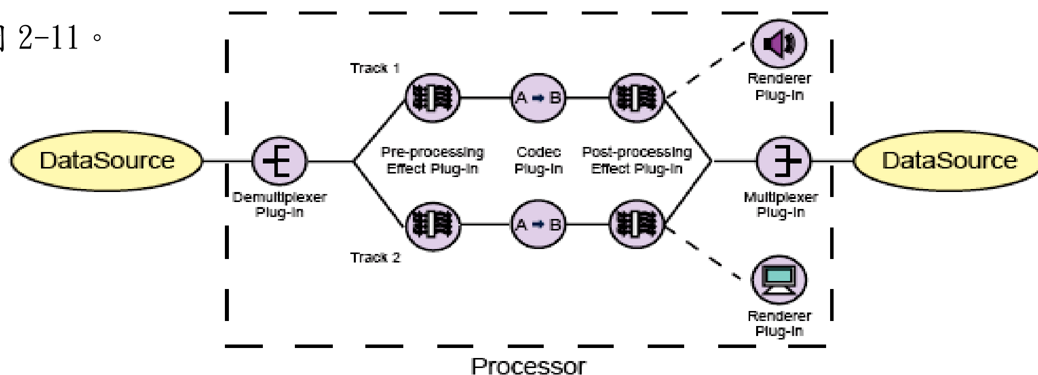


圖 2-11 Process stages

由圖 2-11，processing 元件都為 JMF plug-ins，JMF plug-ins 分別為以下五種:

- Demultiplexing：分析輸入的媒體串流，如 WAV、MPEG 或 QuickTime，將其媒體資料流中個別的 track 抽取出來。
- Effect：對輸入或輸出的媒體資料流中，個別 track 做特效(effect)的演算處理。
- Codec：解壓縮媒體資料，或者將未處理的媒體資料做壓縮處理。
- Multiplexer: 將多個 tracks 合併為單一個輸出資料流，並且做為輸出的 DataSource。
- Renderer：將媒體資料傳送至螢幕或喇叭。

以下三個函式執行結果為建立 Processor，可以依程式設計者的需求分別執行以下函式:

Manager.createProcessor(DataSource source)

Manager.createProcessor(MediaLocator sourceLocator)

Manager.createProcessor (java.net.URL sourceURL)

媒體資料自輸入 Processor，經過各個 processing 元件執行相關資料處理的程序，在這之前必須先了解 Processor 的狀態，如圖 2-12。

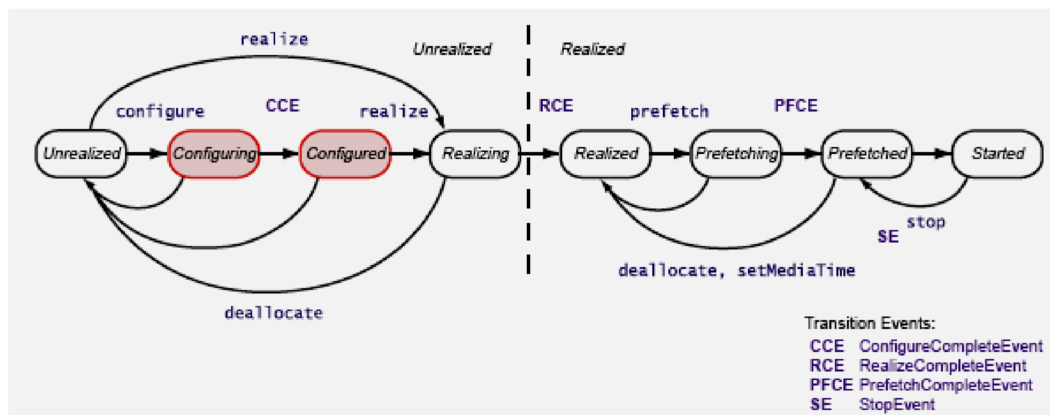


圖 2-12 Processor states

- Unrealized 狀態：剛建立 Processor，意謂著 Processor 與媒體來源並無關聯。
- Configuring 狀態：此時正嘗試連接到 DataSource、 demultiplexes，並且取得輸入的媒體資料格式。
- Configured 狀態：已經連結 DataSource，並且已經計算輸入的媒體資料格式。
- Realizing 狀態：Processor 已計算處理資源(包括網路可下載的資源)的需求，包括 rendering resource。
- Realized 狀態：此時 Processor 已完全建立。

Processor 的輸出可以作為 Player 的輸入，或是直接呈現媒體資料，在此並不討論 Player 與呈現媒體資料(圖 2-11)，因此 Prefetching 狀態之後並不說明。

當 Processor 在 Configured 狀態下，執行 getTrackControl()函式，進而從媒體串流中的 track 得到 TrackControl 物件。這些 TrackControl 物件可以使我們對 media 做一些特別的處理，如選擇那一個(或那些)plug-ins，用來處理音訊或影像 track。可以使用 Processor 的 setContentDescriptor 函式，具體指定 Processor 媒體資料輸出的格式；呼叫 getSupportedContentDescriptors，而得到媒體資料格式的列表。

2.4.2 JMF RTP Architecture

JMF可以傳送和接收RTP streams,主要是定義以下三種套件: javax.media.rtp, javax.media.rtp.event, and javax.media.rtp.rtcp packages.影音來源可為一般檔案或擷取裝置,將影音來源以串流的方式傳送至網際網路或者儲存成一般檔案，如下圖2-13。

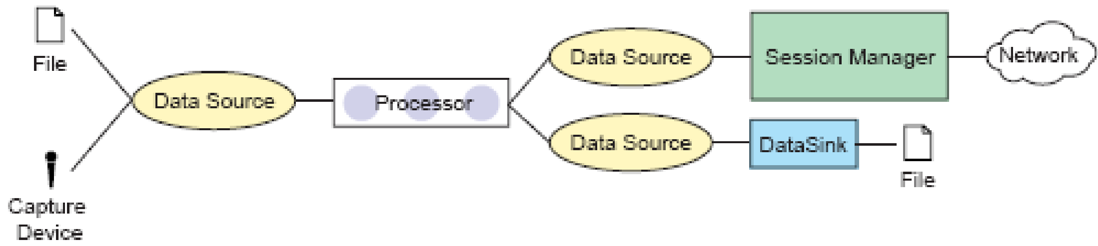


圖 2-13 **RTP transmission**

當接收一串流媒體時，同樣也可以將媒體直接播放或是儲存為一般檔案，如下圖 2-14。

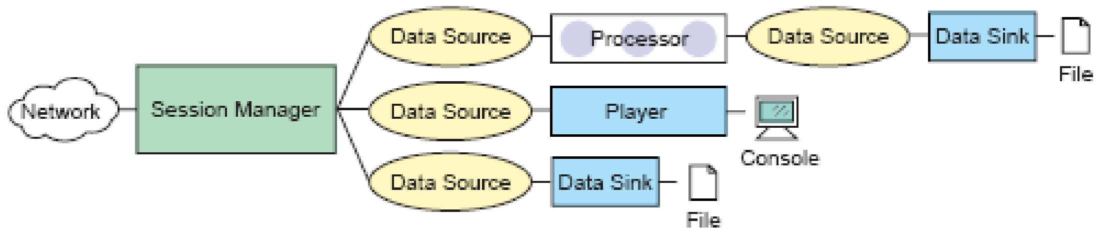


圖 2-14 **RTP reception**

JMF 可以藉由標準 JMF plug-in 機制，延伸支援外加的 RTP formats 和 payloads，如下圖 2-15。

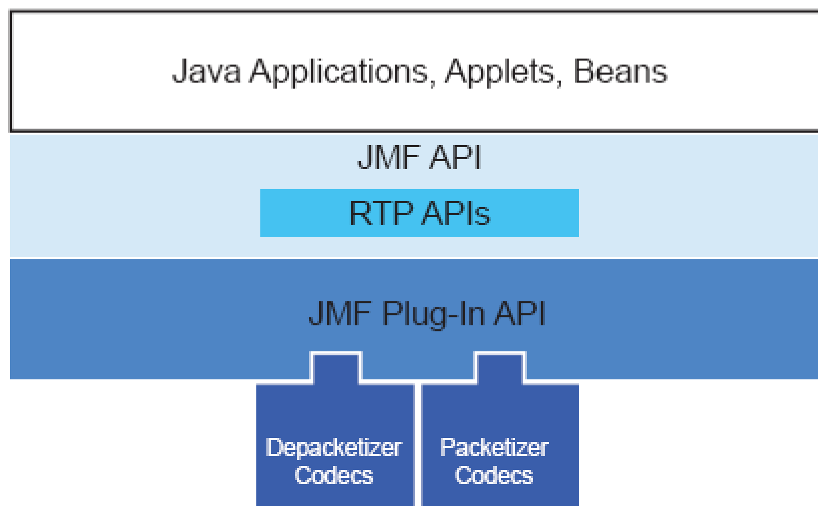


圖 2-15 **JMF RTP architecture**

JMF的SessionManager物件用以協調RTP session，SessionManager借由觀察本地端的成員，維護管理session的狀態，而SessionManager本身也是整個本地端的RTP session代表，也操控RTCP control channel，並且對於sender和

receiver 都有支援RTCP。

在 RTPsession，對於 RTP 資料封包所屬的每個 stream，RTPSession Manager 必須管理一個 RTPStream 的物件。RTPStream 物件為描述在一個 RTP sessions 內的 stream，有兩種形式的 RTPStream: ReceiveStream 和 SendStream，每個 RTPStream 都有一資料來源緩衝器與 RTPStream 相關。以下主要有二種 RTPstreams:

- ReceiveStream: 表示一個已接收的stream是從遠方的接收端傳送過來的。
- SendStream: 表示一串流資料是由 Processor所輸出(DataSource)，而此串流資料將經由網際網路傳送至遠方接收端。根據[5]。

2.5 Java Socket & Real-Time Transport Protocol

2.5.1 Java Socket

Java 語言初期便將支援網路視為必備的要素之一，其擷取 Berkeley Socket 的優點，並加入 Java 跨平臺、物件導向、支援網際網路等考量之下，Java 一推出關於支援網路的套件，則就包括有關於 Socket API 套件:java.net。

Java 發展至今，其支援網路的 API 共有:

- java.net: networking applications。
- java.nio.channels.ServerSocketChannel:new I/O Server Socket Channel。
- java.nio.channels.SocketChannel: new I/O Client Socket Channel。
- javax.net.ssl: networking with Secure Socket Layer(SSL)。
- java.rmi: remote method invocation
- javax.rmi:remote method invocation for IIOP。

接著探討 java.net 套件，其它套件請參考[2]。對 java.net 而言，其套件分別處理(1)主機名稱和 IP 位址;(2)Uniform Resource Locator (URL);(3)TCP 通

訊協定;(4)UDP 通訊協定;(5)網路認證(Authentication);(6)內容處理器(Content Handler)。

(1)主機名稱和 IP 位址:java 使用 java.net.InetAddress 類別來封裝「IP 位址」(其中包括 IPv4 和 IPv6)的概念。當欲查詢某一主機的 IP 位址時，執行 InetAddress.getByName(主機名稱)函式，此函式會以 DNS 來查詢該主機的 IP 位址。

(2)Uniform Resource Locator:處理 URL 時,java 提供 java.net.URL 類別，請參考[2]。

(3)TCP 通訊協定:在 java.net 套件中，其關於支援 Stream Socket 伺服端與用戶端網路應用程式的相關 API 和函式如表格 2-6。

表格 2-6 Java Stream Socket API

	Java API	函式說明
伺服端	java.net.ServerSocket	建立伺服端 Socket 以及設定所使用的 IP 位址和通訊埠。
	accept()	等候與接受自用戶端的連結請求，並且建立與用戶端之連線。
	read()	接收來自用戶端所傳送的資料。
	write()	傳送資料至用戶端。
	close()	關閉 Socket 及與用戶端之通訊連結。
用戶端	java.net.Socket	建立用戶端 Socket，並且嘗試建立與伺服端之連結。
	read()	接收來自伺服端所傳送的資料。
	write()	傳送資料至伺服端。
	close()	關閉 Socket 及與伺服端之通訊連結。

以下為伺服端與用戶端之架構圖(圖 2-16)

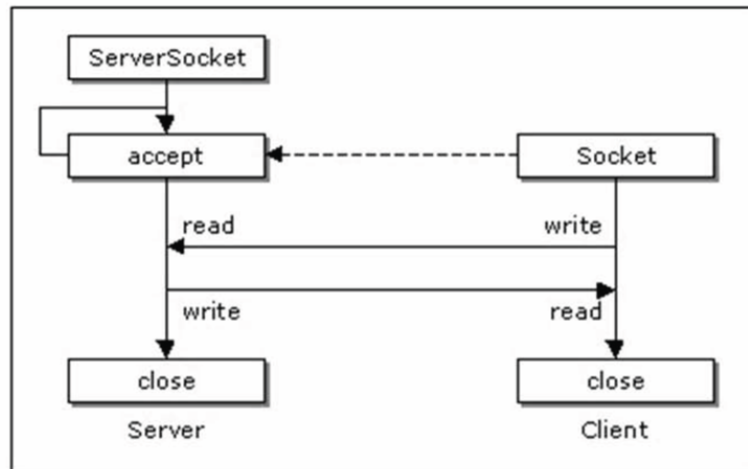


圖 2-16 伺服端與用戶端之架構

(4)UDP 通訊協定: 在 java.net 套件中，其關於支援 Datagram Socket 的相關 API 和函式如表格 2-7:

表格 2-7 Java Datagram Socket API

Java API	說明
DatagramSocket	建立 Datagram Socket。
DatagramePacket	建立 Datagrame Packet。
(DatagramSocket)receive()	接收 Datagrame Packet。
(DatagramSocket)send()	傳送 Datagrame Packet。
Close()	關閉 Datagram Socket。

以下為 Java Datagram Socket 的架構圖(圖 2-17)

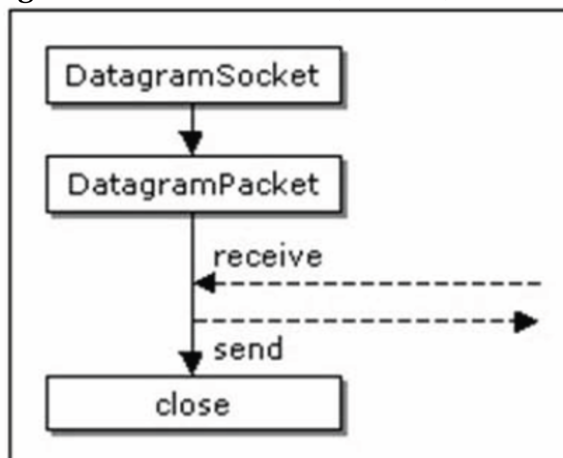


圖 2-17 Java Datagram Socket

TCP 協定是提供可靠的資料傳輸，但由於 TCP 協定的重傳機制使得它並不適合用在即時影音應用。UDP 協定傳輸速度比 TCP 快，雖然 UDP 為不可靠的傳輸協定，但封包丟失或次序錯亂對即時影音而言只會出現雜訊，因此對於即時影音的傳送 UDP 更適合 TCP。

Java 的 UDP 實作分別為兩各類別: DatagramePacket 和 DatagramSocket。DatagramePacket 類別為程式設計者將資料位元組填入 datagram 的 UDP 封包，並且也可自收到的 datagram 取出資料。DatagramSocket 兼具發送與接收 UDP DatagramPacket 的能力。發送資料時，將資料填入 DatagramePacket，接著由 DatagramSocket 送出，反之亦然。

在 UDP 中，與 datagram 有關的資訊都包在封包裡，而 socket 只需知道它應該監聽當地主機的哪一各通訊埠，或是由哪一各通訊埠送出資料，根據[3]。

(5)網路認證(Authentication)、(6)內容處理器(Content Handler) 請參考[2]。

2.5.2 Real-Time Transport Protocol

RTP 提供點對點的網路傳輸服務，適用於即時傳輸的資料，例如各種多媒體資料的傳輸，這些多媒體資料將被包裝成特定格式的 RTP 封包。RTP 架構在 UDP 的上層(如圖 2-18)，將以包裝成特定格式的 RTP 封包傳遞給 UDP，然後藉由 UDP 等網路協定傳送到網際網路上。由於 RTP 對於網路的品質並沒有保證，對此增加控制協定(RTCP)，以達到監控網路的品質。

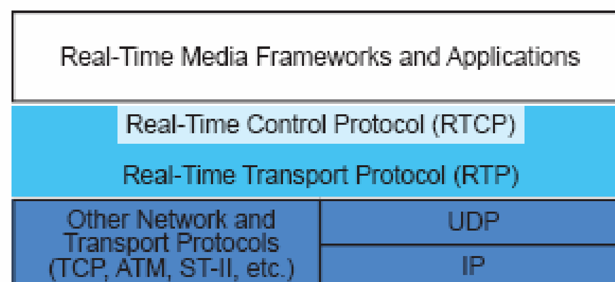


圖 2-18 RTP architecture

2.5.2.1 RTP 檔頭架構

RTP 封包包含一個檔頭以及在檔頭後面的負載資料，RTP 封包的檔頭格式在 RFC3550，根據[4]，中有明確定義，而負載資料格式的切割方式，則根據負載資料不同而有不同的定義，以下為介紹 RTP 封包的檔頭資訊。

The RTP version number (V): 2bits，指出RTP的版本為何，在此RFC3550把此值設定為2。

Padding (P): 1bits，指出封包的結尾有無padding data，padding data可以被用於各種加密演算法。

Extension (X): 1 bit，指出延伸標頭是否存在，若存在則表除了 RTP 固定的檔頭，還放了額外的檔頭資訊。

CSRC Count (CC): 4 bits，RTP packet 是由幾個來源所組成的。

Marker (M): 1 bit.指出此資料流的邊界，當設定為1時表此資料流已達邊界。

Payload Type (PT): 7 bits，是表示為一個索引，此索引指向多媒體設定檔的表，指出payload data的格式。

Sequence Number: 16 bits，此數值指出此封包的位置，數值起始值是隨機產生，隨著封包增加逐次加一。藉由這個數字，接收端可以偵測出遺失封包，以及將順序錯誤的封包進行重新排列。

Timestamp: 32 bits，伺服端會根據取樣時間在每個封包上加上一個時間戳記，使得接收端能得知何時必須處理此封包，當接收端接收的封包已逾時，則此封包將被丟棄。

SSRC: 32 bits，辨別同步來源(Synchronization source)，當此數值為零，表示此payload就為其資料來源，若不為零則表其資料來源為mixer。

CSRC: 最多16項，每項32 bits，指出payload的來源。

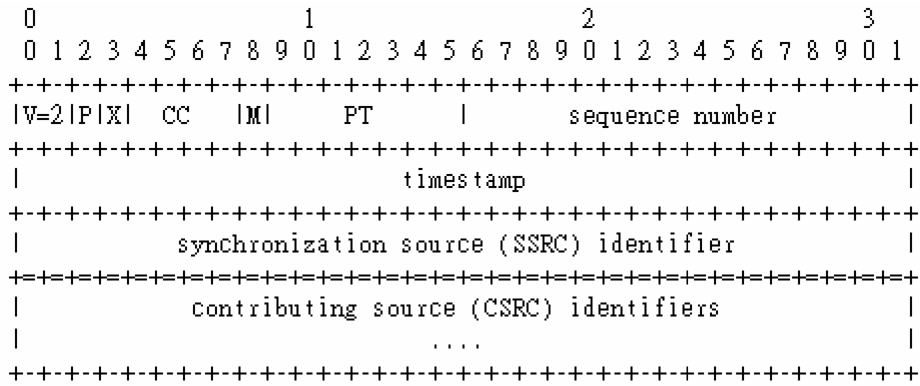


圖 2-19 **RTP data-packet header format.**

第三章 Java多媒體系統動態分析

多媒體資料處理系統，必須由建立數個子系統方塊而組成。其所包括的子系統如下，擷取裝置管理、媒體資料來源、影音來源之分流、影音壓縮處理、影音整合與RTP封包建立和傳送(系統建立之細節請參考第二章)。

本論文依子系統的程式功能建立數個thread，如圖3-1。每個thread彼此間會透過同步機制，互相溝通與合作，傳遞每個子系統所處理完成的媒體資料，並將媒體資料包裝為RTP封包傳遞至接收端。

以下，會依序說明，各個thread的功能、thread的建構流程、thread之間是以何種機制達成溝通與資料分享以及整個系統程式執行的細部運作。

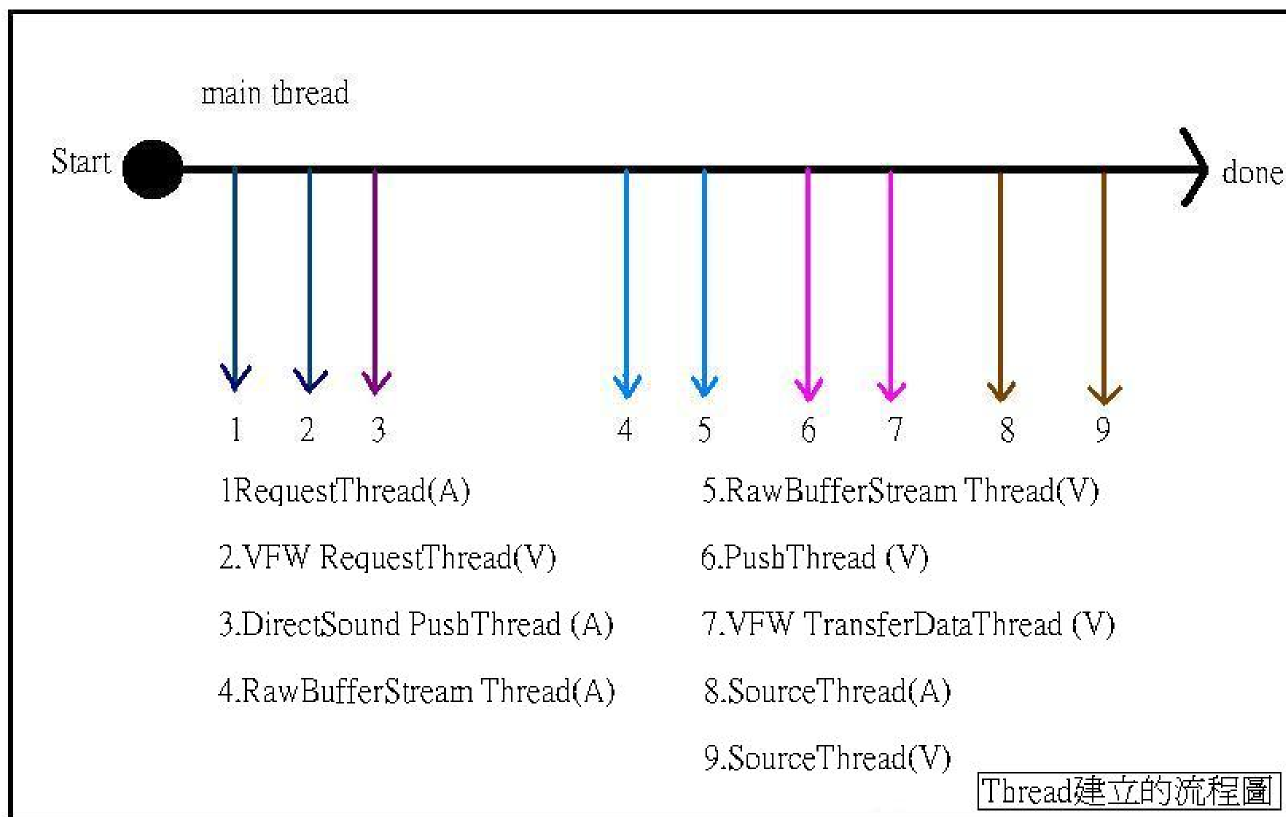


圖3-1 **Thread**建立流程

3.1各個**Thread**的功能介紹

(1)**Audio**方面

RequestThread(A):執行開啟**mike**的程序，或是其它對mike的要求，如關

閉mike。

DirectSound PushThread(A)：將mike所擷取的音訊資料放入子系統：「媒體資料來源」的暫存器，接著將暫存器的音訊資料放入parser內的暫存器。

SourceThread(A):將parser內的音訊資料取出，交由BasicFilterModule做資料壓縮的處理，並且將完成壓縮的資料放入多工器內的暫存器。

RawBufferStreamThread(A):將多工器存的音訊資料交給RTPSession，建立RTP封包，將封包交由接著由下層的UDP送出。

(2)Video方面

VFWRequestThread(V)：執行開啟camera的程序，或是其它對camera的要求，如關閉camera。

PushThread(V):將camera所擷取的影像資料放入子系統:「媒體資料來源」的暫存器。

VFWTransferDataThread(V):將媒體資料來源的暫存器內所存的影像資料寫入parser的暫存器。

SourceThread(V):將parser內的影像資料取出，交由BasicFilterModule做資料壓縮的處理，並且將完成壓縮的資料放入多工器內的暫存器。

RawBufferStream Thread(V): 將多工器所存的影像資料交給RTPSession，建立RTP封包，將封包交由接著由下層的UDP送出。

3.2 Threads建立的流程

由圖3-1Thread建立的流程，以下依序說明其建立流程以及thread所處理的暫存器。

(1)1、2和3 thread的建立

程式Main thread開始執行，建立媒體資料來源，DataSource物件，此物件會包函擷取裝置的路徑和管理影音來源的物件。接著呼叫DataSource物件的connect( )函式，此時會分別建立RequestThread(A)、VFWRequestThread(V)和DirectSoundPushThread (A)。

在建立RequestThread(A)之前，最主要必須為設置存放擷取資料的暫存器大小，暫存器大小的設置與音訊來源有關，程序如下:

1.計算frameSize=取樣位元 X 通道數 / 8

2. 暫存器的大小=取樣率 X frameSize X 預設轉衝器長度(50) / 1000

以表格2-1音訊格式為例:

暫存器的大小=48000 X 4 X 50/1000=9600 (byte)

上例表示每一次擷取的資料量為9600 (byte)。

當建立RequestThread(A)時，RequestThread(A)會進入待命狀態，但由於此thread設立了request旗標，其預設為0，唯有當request旗標大於0時，run( )函式裡，判斷式條件成立才能對mike改變狀態。但由於request旗標此時為0，因此，此thread雖為待命狀態，只有當收到旗標狀態變更的通知，並且判斷式條件成立，此thread才能對mike改變狀態。在此會先建立DirectSound PushThread(A)，由於未呼叫start( )函式，此時為初始狀態。

VFW RequestThread(V)設立vfwRequest旗標，其預設為-1，唯有當vfwRequest旗標大於或等於0時，此thread才會進入到執行狀態，但由於vfwRequest旗標此時為-1，因此此thread會放出執行權，並且進入封鎖狀態，但由於它呼叫為sleep( )函式，因此當設立的時間到時，它會再次醒來，但若還有其它thread也在競爭執行權時，就可能超過預定的等待時間才醒來。醒來後同樣會再次檢查vfwRequest旗標。在此設置此暫存器的大小是依影像大小而決定，以表格2-1影像格式為例:每次擷取的資料量為28800(byte)。

(2)4和5thread的建立

Main thread呼叫realizeProcessor( )函式，會建立RealizeWorkThread物件，此thread會進入執行狀態，執行以下程序:呼叫doRealize( )函式，當呼叫此函式時，會建立BasicFilterModule物件(處理媒體資料壓縮)和呼叫connectMux( )函式，建立RTPSyncBufferMux物件(多工器)，初始多工器的輸入端時，會建立兩個RawBufferSourceStream物件，此物件是媒體(A/V)輸入資料存放的地方。當建立RawBufferSourceStream物件的同時，會依序建立RawBufferStreamThread(A)和RawBufferStreamThread(V)，此兩個thread都會進入初始狀態，直到Main thread建立了RTPSessionMgr物件和SendStream物件(所存放的媒體資料將會封成RTP封包)，執行SendStream.start( )函式時(請參考2.3.4節)，此兩個thread才會呼叫start()函式才會進入待命狀態，。

由於RealizeWorkThread的run( )函式並非迴圈，當上面程序都執行完成後，此thread會進入結束狀態，此thread會佔用記憶體位置，Java提供自動回收廢棄不用的記憶體(automatic garbage collection),也提供記憶體回收執行緒(garbage-collector thread),會自動回收程式不再使用的記憶體。

(3)6、7、8和9 thread的建立

最後必須要求實際的媒體資料輸入，因此會建立PrefetchWorkThread，此thread會進入執行狀態，接著呼叫doPrefetch( )函式，接著執行以下程序:BasicSourceModule.doStart( )，其會呼叫RawBufferParser.start( )函式，接著會呼叫PushBufferStream陣列內的VFWSourceStream.start( )函式與DirectSoundStream.start( )函式。

當呼叫VFWSourceStream.start( )函式時，則會建立PushThread(V)，此thread會進入待命狀態，當thread取得執行權，進入執行狀態，開始執行時會先睡0.01秒，醒來後會呼叫yield( )函式，此函式目的是要達到公平的排程。緊接著會建立VFWTransferDataThread(V)，此thread會進入執行狀態，會先檢查

bufferQ(在此bufferQ泛指儲存資料的暫存器)是否存有資料，若無資料，則會進入鎖定狀態，等待0.25秒，醒來後一樣會再檢查bufferQ。建立上面兩個thread的同時，會送出vfwRequest=1，改變VFW RequestThread(V)旗標的狀態，此thread會開啟camera。

當呼叫DirectSoundStream.start( )函式，會先送出request旗標為3至RequestThread(A)，此thread則會開啟mike。接著會呼叫DirectSound PushThread(A).start()，此thread會進入待命狀態。

接著會依據有多少影音tracks而建立相同數目的SourceThread，在此會有SourceThread(A)和SourceThread(V)，並且都會進入等待狀態，當thread取得執行權進入執行狀態，當檢查bufferQ為不可讀時，則此thread會進入封鎖狀態，直到此thread收到解開封鎖的通告，此thread才會再進入執行狀態。

由於PrefetchWorkThread的run函式並非迴圈，當上面程序都執行完成後，此thread會進入結束狀態。

3.3 Thread之間的資料分享與溝通

本論文所使用的排程機制為3.6.5節所提:自定排程，對各個thread用wait( )函式保留執行權。處理媒體資料的各個子系統，當各自要處理媒體資料時，都會將媒體資料存在各自的暫存器，因此thread就必須做為每個暫存器之間的橋樑，負責將媒體資料自暫存器讀出，或是將媒體資料放入暫存器。

子系統內的暫存器，其所扮演的角色就是thread能否執行的條件式，若thread無法執行任務時，子系統內的暫存器就會扮演管理thread的角色，直到收到此thread可執行條件成立的通告，就會解開對此thread的管理，而此thread就會鎖住此暫存器，而執行資料讀出或放入的任務。

由上可知，本論文是利用子系統內的暫存器，做為每個thread互相制衡的機制，同時每個子系統也必須各自完成，處理媒體的任務。以上Java thread的相

關知識請參考3.6節。

3.3.1 子系統介紹和Thread所屬之類別物件

(1) 子系統介紹

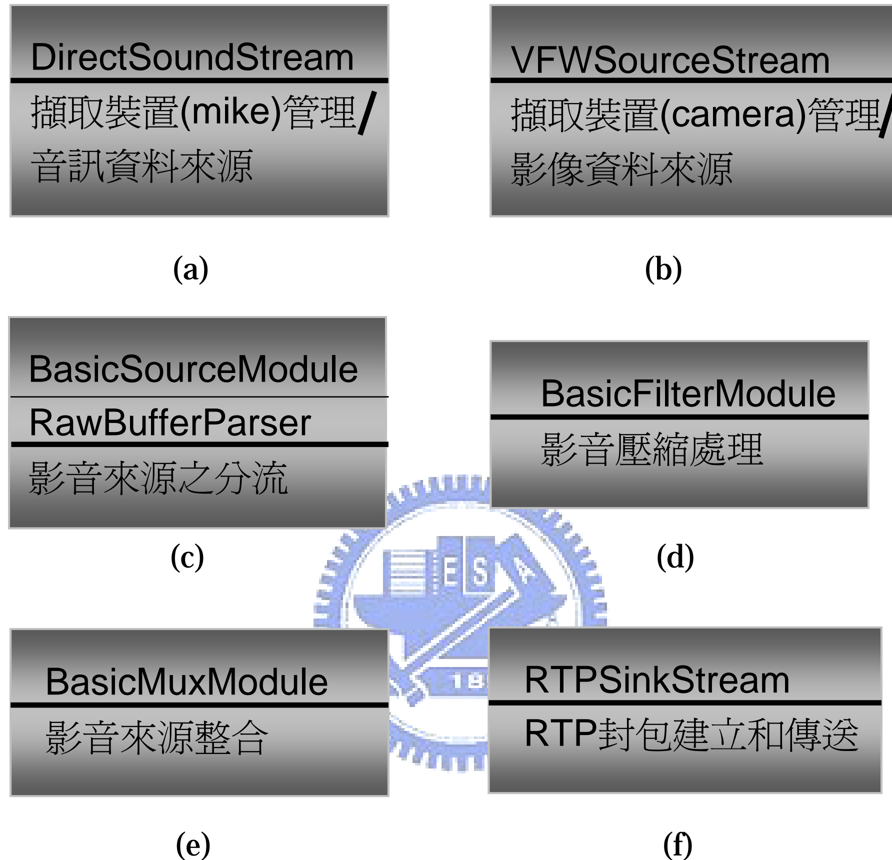


圖3-2(a)音訊擷取裝置管理、音訊資料來源；(b)影像擷取裝置管理、影像資料來源；(c)影音來源之分流;(d)影音壓縮處理; (e)影音來源整合;(f) RTP封包建立和傳送。

以上為各個子系統的實際物件，黑色實線之下方表示其物件的功能。

(2) Thread所屬之類別物件

在"3.6 Java thread"會提到，同一個類別可產生多個thread，圖3-3為介紹每個thread是包含在哪些類別，並且說明子系統所擁有的暫存器內容。

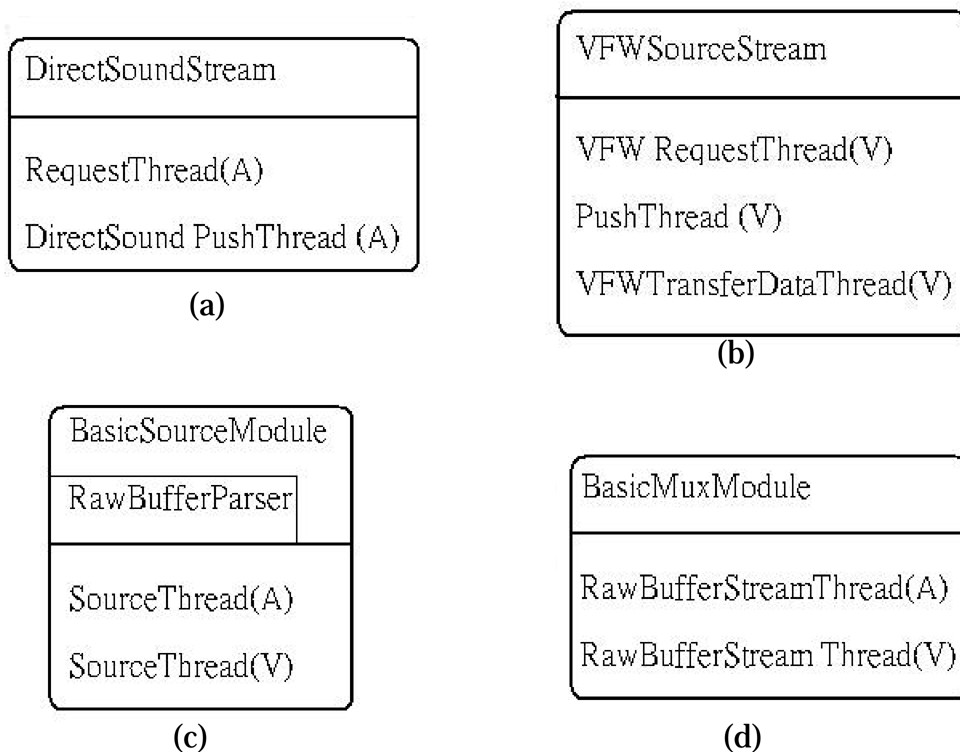


圖3-3 (a)音訊來源(b)影像來源(c)影音分流(d)多工器輸入來源

在3.2節，有說明每一次mike所抓取的資料大小，在DirectSoundStream物件內會有一bufferQ物件的暫存器，且為FIFO的特性，此暫存器擁有一個queue，queue儲存大小與3.2節所計算的資料量大小相同。VFWSourceStream物件內會有一bufferQ物件的暫存器，且為FIFO的特性，此暫存器擁有八個queue，每個queue儲存大小與擷取的影像資料量有關，如表格2-1其大小為28800(byte)。

在RawBufferParser物件內的bufferQ物件的暫存器所擁有的queue的數量：音訊方面為1，影像為1，兩者大小設置都與前者相同。BasicMuxModule物件內的bufferQ物件的暫存器同樣具有FIFO的特性，其所擁有的queue的數量，影音皆為5，每個queue儲存大小皆為1456(byte)。

媒體資料來源DataSource物件會包含DirectSoundStream物件和VFWSourceStream物件。VFWSourceStream物件會將擷取到的影像資料放入暫存器中，其過程為VFWRequestThread(V)負責管理camera，當

VFWRequestThread (V)開啟camera，PushThread(V)將擷取的影像資料存入暫存器中。VFWTransferDataThread(V)會將暫存器中的影像資料取出，接著放入RawBufferParser物件的暫存器，在以上過程中，我們會發現暫存器是由後兩個thread所共用的，因此當thread要放入或讀出資料時，都必須注意資料同步的問題。

由上圖3-3，除了VFWSourceStream物件內包含數個thread，要注意資料同步的問題之外，每個物件之間，資料的傳遞則是靠著thread之間的溝通，主要是每個物件都有提供一個機制能讓thread成為等待區，此機制能夠幫助thread間的通訊。

3.3.2 Thread的設計原理與制衡

依thread所處理的媒體資料特性，分別以處理音訊和影像資料的thread來加以探討。Java虛擬機對作業系統來說，也是一支程式，Java虛擬機會自作業系統取得執行權，接著將執行權分給各個thread使用。在此系統中，所有的thread的run()函式都是無窮迴圈，除非main thread完全結束，否則各個thread都會依排程機制繼續執行。

在此，我們必須先了解一件事，在此的9個thread(包含處理影像和音訊)都為無窮迴圈，每一個thread都是相同的優先度，在任何時間每個thread都是有權利競爭執行權，除了處在封鎖狀態的thread是無法競爭。Thread會進入封鎖狀態，是因未滿足某一條件，在此的條件只有兩種，可否放入資料:「判斷記錄暫存器剩餘空間的參數值若為0，表示暫存器不允許放入資料」和可否讀出資料:「判斷紀錄暫存器資料量的參數值若為0，表示暫存器不允許讀出資料」。

3.3.2.1 處理影像資料

此小節說明處理影像 Thread 的設計原理與制衡機制。在圖 3-4 中，處理影

像資料的 threads 會執行各自的任務，所有箭頭的起始點和終點表示為各個 thread 所必須要處理的物件，以及物件內的暫存器，其中 bufferQ1 有 8 個 queue，bufferQ2 有 1 個 queue，前兩者的 queue 都為儲存一張照片的大小。bufferQ3 有 5 個 queue，每個 queue 都為儲存一預設 RTP 封包的大小。

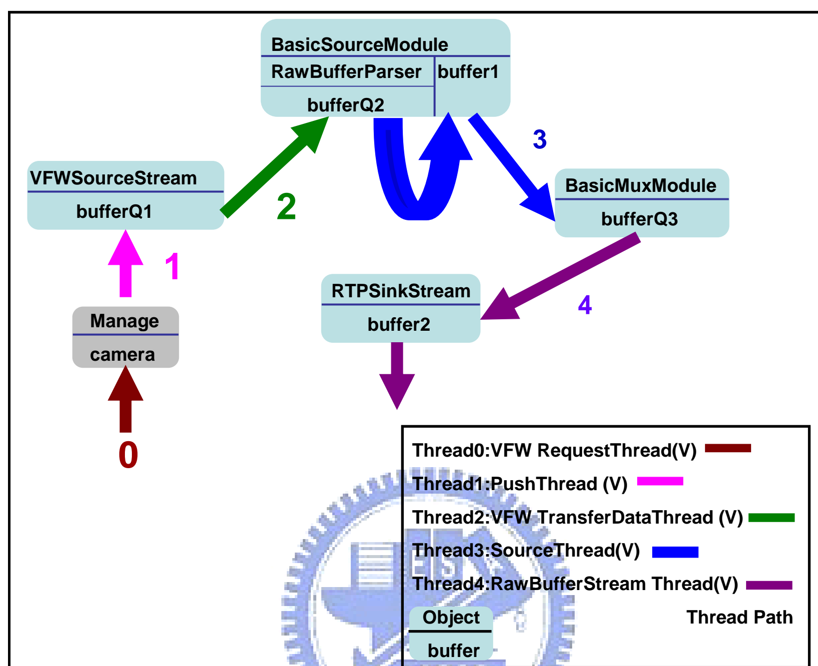


圖 3-4 Thread Path(V)

(1) Thread 的設計原理

以下會針對每一個 thread 的設計概念做說明，當有對 bufferQ1、bufferQ2 或 bufferQ3 作處理時，都會啟動同步機制，此 3 個物件會宣告為 synchronized(請參考 3.6.3~3.6.6 節)。

Thread0：負責管理 camera，如開啟 camera。除非有其它 thread 要求它改變狀態，否則當它取得執行權時，並不會做任何程序就會再度回到待命狀態。

Thread1：負責從 camera 取得影像資料，並存入 bufferQ1。

由於擷取影像資料的時間點不能太近，並且希望盡量達到公平的排程，因此使用 sleep(time)函式，使得 Thread1 休眠數豪秒後再起來執行，並且使用 yield()函式，若 Thread1 此次已執行過，下一次的執行機會能讓給其它的 thread，以上流程如圖 3-5。

接著必須檢查bufferQ1可否放入資料，當無法放入資料時，表示已有8張照片未處理，因此會清除在bufferQ1內最早放入的照片。接著將擷取的影像資料放入一暫存器buffer，再將buffer內的資料放入bufferQ1，此時會執行bufferQ1.notifyAll()函式，此函式主要為通知Thread2隨時可取走資料，接著將記錄buffer資料量的參數設為0(接下說明相同的程序時都稱之為「清空暫存器」)，為下一次的影像資料做準備。以上流程如圖3-5。

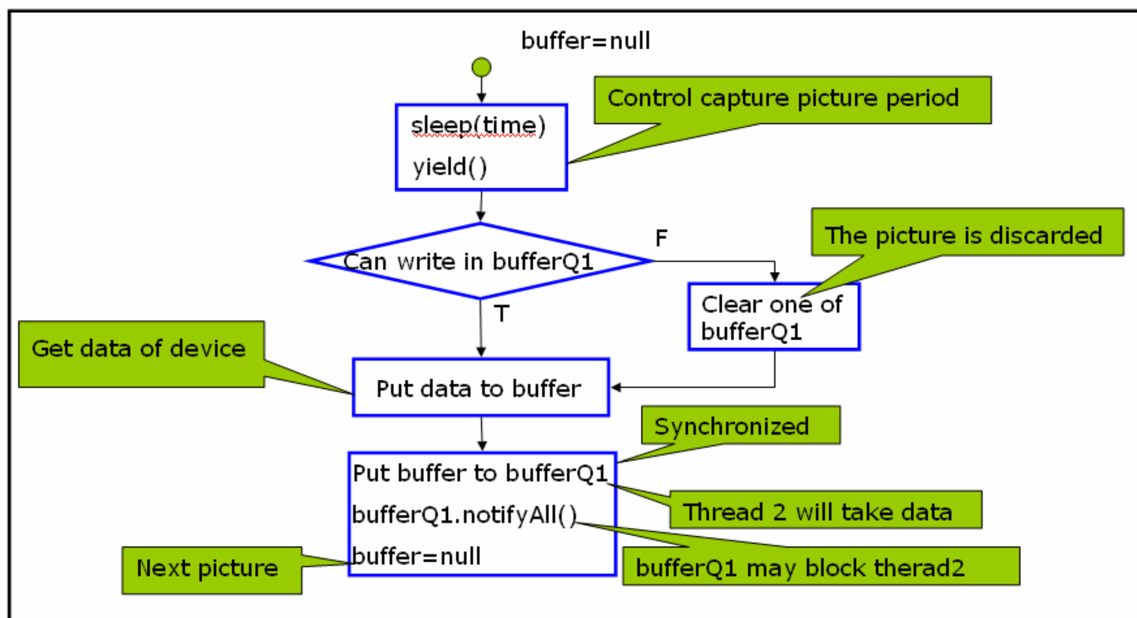


圖3-5 PushThread(V)

Thread2:負責將bufferQ1內的資料搬運至bufferQ2。

首先必須要確定兩件事:1. Thread2可以取得影像資料，因此會檢查bufferQ1可否讀出資料，若無法取得影像資料Thread2就會在此等待影像資料。當確定可以取得影像資料時，必須確定另一件事:2. Thread2可否將資料放入bufferQ2，因此會檢查bufferQ2可否放入資料，當判定無法放入資料時，Thread2會在此等待Thread3將bufferQ2內的資料取走。

當上述兩件事都成立時，才會開始執行真正資料搬運的動作。首先，會先宣告一暫存器buffer，將bufferQ1內的資料放入buffer，並且清空bufferQ1所存放

此資料的queue。接著再將buffer內的資料放入bufferQ2，並執行bufferQ2.notifyAll()函式，通知Thread3隨時可取走資料。以上流程如圖3-6。

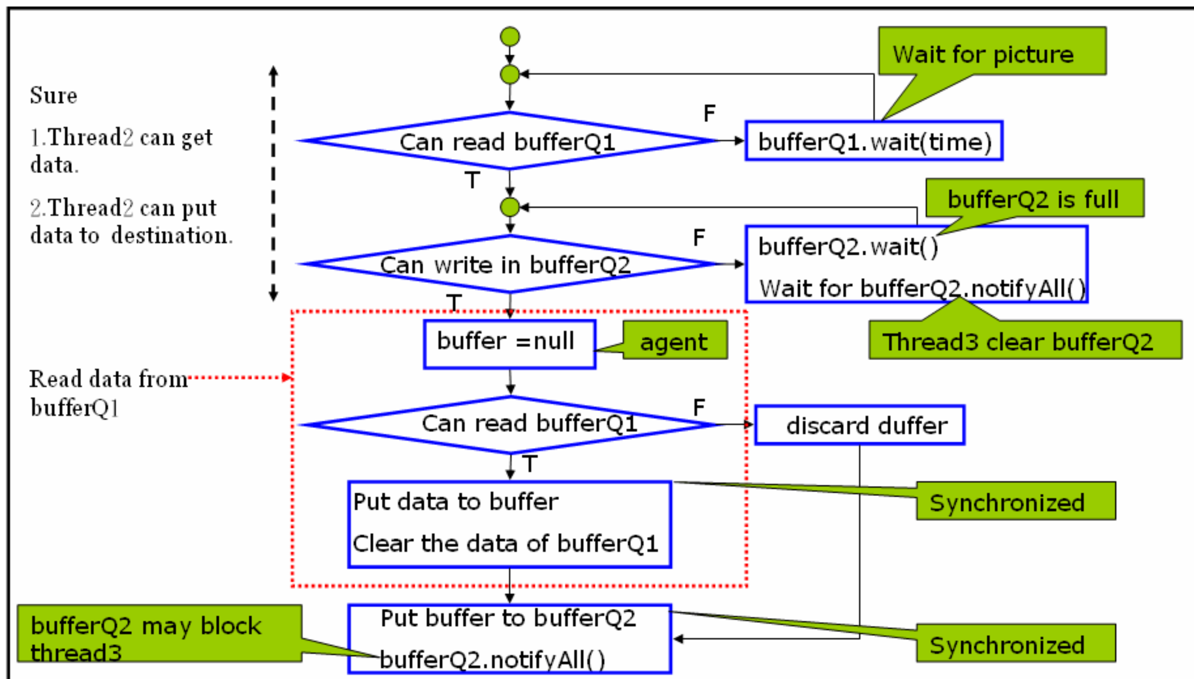


圖3-6 VFWTransferDataThread(V)

Thread3: 負責自bufferQ2讀出資料後，對取出的資料做壓縮的處理，同時切割為數個封包，並且依序放入bufferQ3。

首先必須確定Thread3能否取得影像資料，因此檢查bufferQ2可否讀出資料，當無法自bufferQ2讀出資料時，Thread3會等待Thread2將影像資料放入bufferQ2。

當Thread3可以自bufferQ2讀取資料時，宣告一暫存器buffer1，將bufferQ2的資料放入buffer1，接著清空bufferQ2並通知Thread2隨時可放入新的影像資料。將資料暫時存入buffer1是為了要對資料壓縮的處理做準備，當buffer1的資料壓縮好並切割為封包大小後放入buffer1‘，接著buffer1‘所存的封包必須要逐一放入bufferQ3。

因此必須檢查bufferQ3可否放入資料，當無法放入資料時，Thread3會等待Thread4取出bufferQ3內的封包。當Thread3可以將封包放入bufferQ3時，會宣告一暫存器buffer，接著將buffer1‘內的封包放入buffer，再將buffer內的封包逐

一放入bufferQ3，並且告知Thread4隨時可將封包取出。

最後Thread3會檢查自buffer送出的封包是不是為最後一個封包，若是最後一個封包，則Thread3會等待一段時間，此做法是希望再壓縮下一張照片前，Thread4已將此次照片相關的封包都發送自網際網路，當Thread4發送出最後一個封包時就會通知Thread3可以壓縮下一張照片。以上流程如圖3-7。

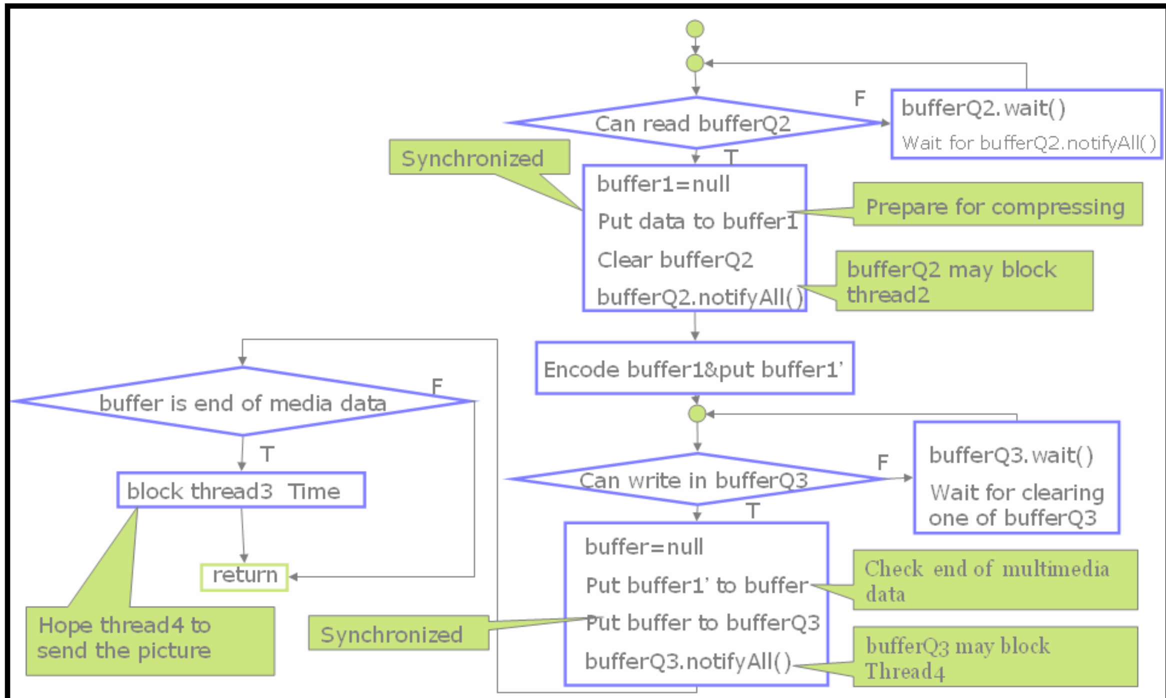


圖3-7 SourceThread(V)

Thread4: 負責將bufferQ3內的封包一一讀出，建構為RTP封包，並交給Socket發送。

首先Thread4必須確定是否有封包要發送，因此必須檢查bufferQ3可否讀出資料，若判定無法讀出資料，Thread4則會等待Thread3將封包放入bufferQ3。當Thread4可以自bufferQ3讀出封包時，宣告一暫存器buffer，接將bufferQ3內的封包放入buffer，接著檢查buffer內所存的是不是為最後一個封包，若是最後一個封包，Thread4會通知Thread3可以壓縮下一張照片。最後清空bufferQ3內存放此次封包的queue。以上流程如圖3-8。

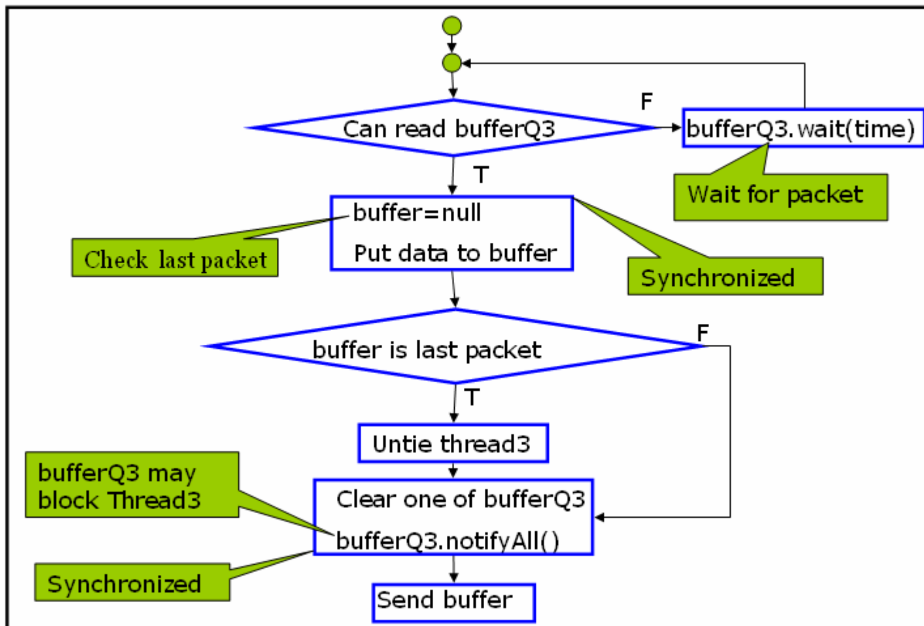


圖3-8 RawBufferStreamThread(V)

(2) Thread制衡機制

由上述各個 thread 的執行概念，下圖 3-9 為暫存器可能管理的 thread 之整理。bufferQ1 可能管理 VFWTransferDataThread(V)，藉由封鎖時間的結束，VFWTransferDataThread(V) 可自動解開 bufferQ1 的封鎖，或是由 PushThread(V) 將資料放入 bufferQ1 時，通知 bufferQ1 解除 VFWTransferDataThread(V) 的封鎖，VFWTransferDataThread(V) 隨時可自 bufferQ1 讀取資料。

bufferQ2 可能管理 VFWTransferDataThread(V) 或 SourceThead(V)，也意味著此兩個 thread 同時是藉由 bufferQ2 來溝通。當 VFWTransferDataThread(V) 無法將資料放入 bufferQ2 時，此 thread 會被 bufferQ2 封鎖，直到 SourceThead(V) 通知 bufferQ2 解除對 VFWTransferDataThread(V) 的封鎖時，VFWTransferDataThread(V)就會得知可以將資料放入 bufferQ2 的通知。反之亦然。

bufferQ3 可能管理 SourceThead(V) 或 RawBufferStreamThread(V) ，也意味著此兩個 thread 同時是藉由 bufferQ3 來溝通。當 SourceThead(V)無法將資料放入 bufferQ3 時，此 thread 會被 bufferQ3 封鎖，直到

RawBufferStreamThread(V)通知 bufferQ3 解除對 SourceThead(V)的封鎖時，SourceThead(V)就會得知可以將資料放入 bufferQ2 的通知。

當 RawBufferStreamThread(V)無法自 bufferQ3 取得封包時，會被 bufferQ3 鎖住，此 thread 可藉由封鎖時間結束而自動結束封鎖，或是 SourceThead(V)將封包放入 bufferQ3 時，會通知 bufferQ3 解除對 RawBufferStreamThread(V)的封鎖。

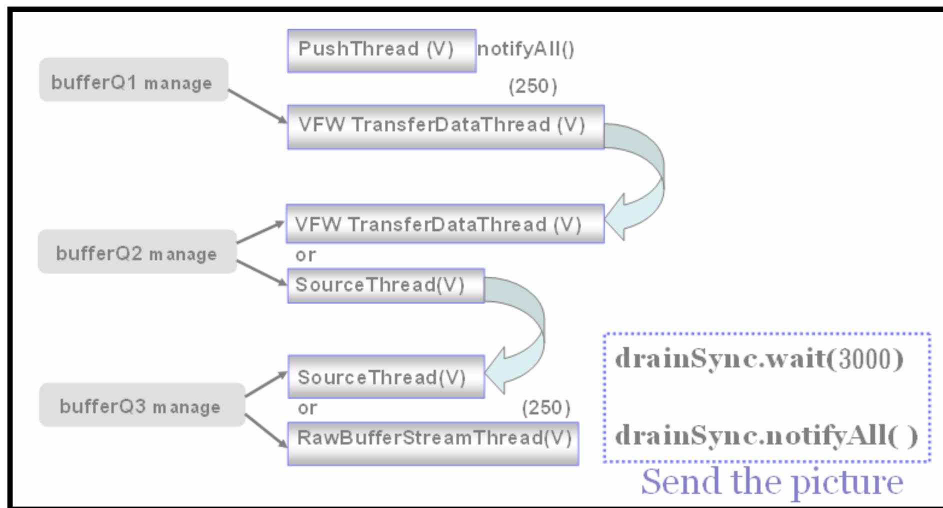


圖 3-9 buffer manage(V)

由 Thread 的設計原理可知 SourceThead(V)和 RawBufferStreamThread(V)之間還外加有一控制機制。當 SourceThead(V)壓縮並送出最後一個封包時，必須進入封鎖狀態，等待 RawBufferStreamThread(V)將 bufferQ3 內的最後一個封包發送出去時，SourceThead(V)就會解開封鎖繼續壓縮下一張照片。如圖 3-9 的右下方。

(3)暫存器狀態判斷機制

由前，對於Thread間的制衡機制的介紹，可知必須要有相關機制加以判斷暫存器是否可以被放入或是讀取資料。當要判斷可否自暫存器讀出資料時，呼叫 canRead( )函式，此函式會檢查記錄暫存器資料量的參數是否大於零，當大於零時，會回傳一true的布林值，表示此暫存器內存有資料。當要判斷可否放入資料

時，呼叫canWrite()函式，此函式會檢查記錄暫存器剩餘空間的參數值是否為零，當大於零時，會回傳一true的布林值，表暫存器還可放入新資料。

3.3.2.2 處理音訊資料

此小節說明處理音訊 Thread 的設計原理與制衡機制。在圖 3-9 中，處理音訊資料的 threads 會執行各自的任務，所有箭頭的起始點和終點表示為各個 thread 所必須要處理的物件，以及物件內的暫存器，其中 bufferQ1 有 1queue，bufferQ2 有 1queue。bufferQ3 有 5 個 queue，每個 queue 都為儲存一預設 RTP 封包的大小。

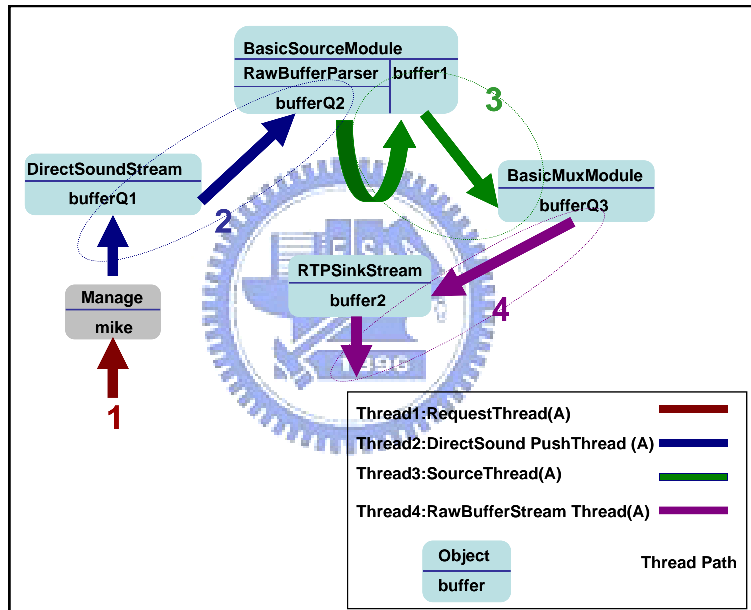


圖3-10 Thread Path(A)

(1) Thread 的設計原理

以下會針對每一個 thread 的設計概念做說明，當有對 bufferQ1、bufferQ2 或 bufferQ3 作處理時，都會啟動同步機制，此 3 個物件會宣告為 synchronized(請參考 3.6.3~3.6.6 節)。由於在影像處理時有說明 Thread3 和 Thread4，因此不再說明此兩個 thread。

Thread1：為負責開啟 mike，除非有其它 thread 要求它改變狀態，否則當它取得執行權時，並不會做任何程序就會再度回到待命狀態。

Thread2：自 mike 取得音訊資料並放入 bufferQ1，接著將 bufferQ1 所存之音訊

資料放入 bufferQ2。

由圖 3-10 中，Thread2 要將擷取的音訊資料放入 bufferQ1 時，會宣告 bufferQ1 為 synchronized 接著取得 bufferQ1 的 token，並且檢查 bufferQ1 可否放入資料，當無法放入資料時會將記錄 bufferQ1 資料量的參數設為 0(接下說明相同的程序時都稱之為「清空暫存器」)。接著會將音訊資料放入 bufferQ1。

接著必須先確定 bufferQ2 允許放入資料，Thread2 才會執行搬運資料的動作。因此必須檢查 bufferQ2 可否放入資料，當無法放入資料時必須等待 Thread3 取出 bufferQ2 內的資料，所以必須執行 bufferQ2.wait()函式，Thread2 會進入封鎖狀態直到 Thread3 通知 Thread2 可放入資料時，所執行的 bufferQ2. notifyAll()函式來解除 Thread2 的封鎖狀態。

當確定可以將資料放入 bufferQ2 時，會宣告一 buffer 的暫存器，做為自 bufferQ1 傳遞資料至 bufferQ2 的仲介。當 bufferQ1 的資料放入 buffer 內時，接著會執行清空 bufferQ1，最後將 buffer 內的資料放入 bufferQ2。以上流程如圖 3-11。

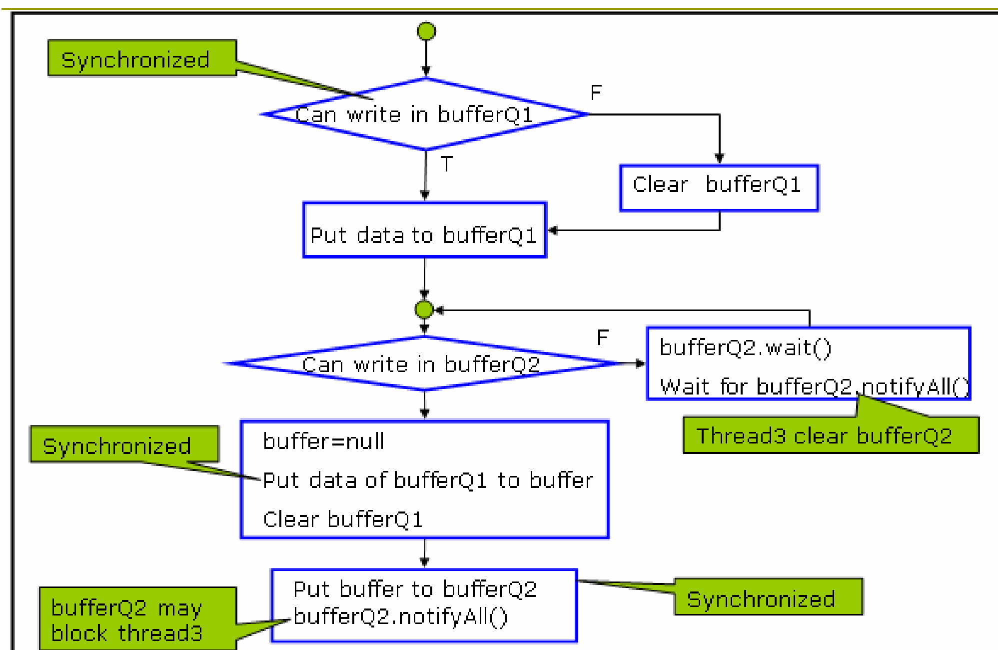


圖 3-11 DirectSoundPushThread(A)

(2) Thread制衡機制

由上各個 thread 互相牽制的情形，圖 3-12 為暫存器可能管理的 thread 之整理。

其原理與圖 3-9 雷同。

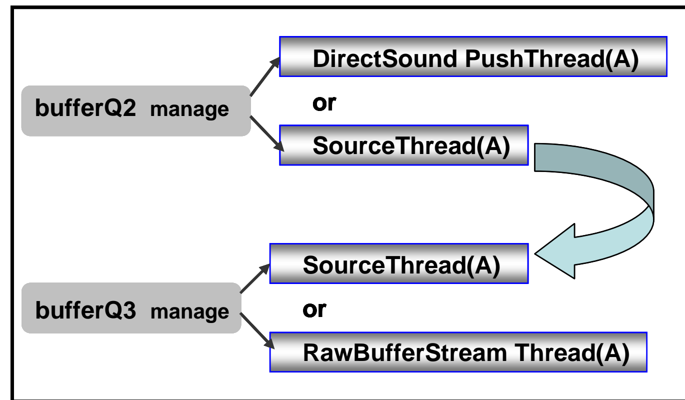


圖 3-12 buffer manage(A)

3.4 系統程式執行的細部運作

此小節會說明整個處理影像和音訊資料的虛擬碼，如圖3-13和3-18。

3.4.1 Video資料的處理流程

Manage
camera

VFWSourceStream
bufferQ1

BasicSourceModule
RawBufferParser
buffer1
bufferQ2

BasicMuxModule
bufferQ3

RTPSinkStream
buffer2

1
2
1
2
1
2

buffer = null (default value)
sleep(10)
yield( )
Can write in bufferQ1 ?
If no
Clear one of bufferQ1.

checkDataAllocation(buffer)
Put buffer to bufferQ1.
bufferQ1.notifyAll()
clear buffer.

Can read bufferQ1 ?
If no
bufferQ1.wait(250)
flag:doneReading = false
If can read bufferQ1
pushData( )

Can write in bufferQ2?
If no
bufferQ2.wait()
buffer=null
read(buffer)
{if Can read bufferQ1 ?
buffer=bufferQ1.read()
clear bufferQ1
else
buffer.setDiscard(true)}
doneReading = true
Put buffer to bufferQ2.
bufferQ2.notifyAll()

Can read bufferQ2 ?
If no, bufferQ2.wait( )
else{ buffer1=null
buffer1=bufferQ2.read( )
clear bufferQ2
bufferQ2.notifyAll( )}
Encode buffer1
Multiplexer.process
(buffer1',index)
→prrocess(buffer1')

can write in bufferQ3 ?
If no
bufferQ3.wait( )
else
buffer=null
buffer=setData
(buffer1'.getData( ))
Put buffer to bufferQ3.
bufferQ3.notifyAll( )

If buffer1' isEOM
drainSync.wait(3000)

If cann't read bufferQ3 ?
bufferQ3.wait(250)
else
transferData( )
→some data ready to be sent over.

If cann't read bufferQ3?
bufferQ3.wait( )
else
buffer2 =bufferQ3.read( )
If buffer2 isEOM
drainSync. notifyAll()
Clear bufferQ3
bufferQ3.notifyAll( )

buffer2→RTPPackage →UDP

VFW RequestThread(V)
PushThread (V)
VFW TransferDataThread (V)
SourceThread(V)
RawBufferStream Thread(V)
Object
buffer
Thread Path

圖 3-13 系統運作(V)

圖 3-13 為 Video 資料的處理流程，接下來會分四個部分說明：(1)擷取影像資料；(2)抽取影像資料放入 Parser；(3)影像資料壓縮處理並送至多工器；(4) RTP 封包建立與發送之前置處理。

(1)擷取影像資料

當處理影像資料時，假設 VFWRequestThread(V)已將 camera 開啟，接著 PushThread(V)拿到執行權時，會先睡 0.01 秒，當它再度拿到執行權時會呼叫 yield( )函式。在此函式意謂著 PushThread(V)即可將執行權讓給其他的可執行狀態的 thread。排程器會從可執行狀態的 thread 中，選出一個 thread 來執行，此函式的目的是為了對多個 thread 盡可能的達到公平的排程，把排程的機會給等待中的 thread。

PushThread(V)內會宣告一 buffer，接著會檢查 bufferQ1 可否放入資料，當檢查結果為不可放入資料時，表示已有逾時的資料未處理，將忽略 bufferQ1 內最早存入影像資料的 queue，則會將記錄 queue 資料量的參數設為 0(接下說明相同的程序時都稱之為「清空暫存器」)，接著執行 checkDataAllocation()函式，將此次擷取的影像資料存入 buffer。再將 buffer 內的資料放入 bufferQ1，並且通知 VFWTransferDataThread(V)隨時可讀取出資料，最後清空 buffer，為下一張照片作準備。以上流程如圖 3-14。

```
buffer = null     (default value)
sleep(10)
yield( )
Can write in bufferQ1 ?
If no
Clear one of   bufferQ1.

checkDataAllocation(buffer)
Put buffer to bufferQ1.
bufferQ1.notifyAll()
clear buffer.
```

圖 3-14 擷取影像資料

(2)抽取影像資料放入 Parser

首先 VFWTransferDataThread(V)會先檢查可否讀取 bufferQ1 內的影像資資料，若無法讀取 bufferQ1 內的影像資資料時，VFWTransferDataThread(V)會進入封鎖狀態，預設在 0.25 秒後解除封鎖，或是當 PushThread(V)將影像資料放入 bufferQ1 時也會自動解除封鎖。當檢查 bufferQ1 為允許被讀取資料，接著會設置一 doneReading 的旗標，此旗標所代表為 bufferQ1 內的資料是否已被讀取。

VFWTransferDataThread(V)會執行 pushData( )函式，此函式會詢問是否能將影像資料放入 RawBufferParser 內的 bufferQ2。當可以將影像資料放入 bufferQ2 時，VFWTransferDataThread(V)內會宣告一暫存器 buffer，接著執行 read(buffer)函式，read(buffer)函式同樣會再次檢查可否讀取 bufferQ1 內的影像資料，當可以讀取資料時，將 bufferQ1 的資料存入 buffer，接著清空 bufferQ1 內儲存此次影像資料的 queue。doneReading 的旗標將會改為已讀取資料的狀態。而 bufferQ1 無法讀取資料時，就直接忽視此次的資料存取，同樣 doneReading 的旗標將會改為已讀取資料的狀態。將記錄 buffer discard 狀態的旗標設為 1。

最後將 buffer 內的資料放入 bufferQ2，並通知 SourceThread(A)隨時可以取得影像資料。此時已完成將影像資料放入 RawBufferParser 內的 bufferQ2。以上流程如圖 3-15。

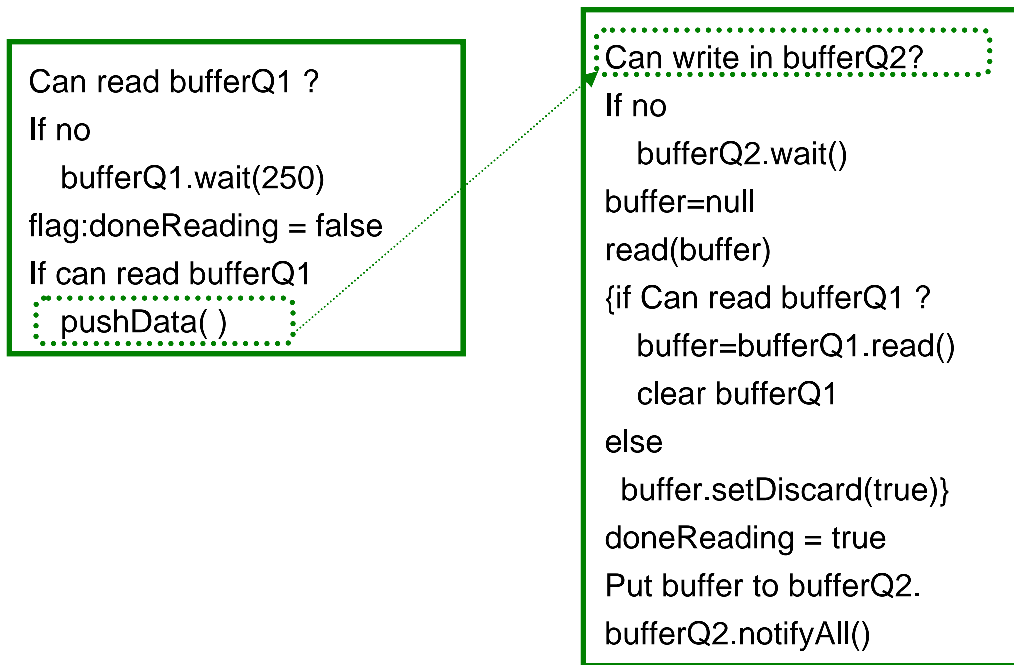


圖 3-15 抽取影像資料

(3)影像資料壓縮處理並送至多工器

SourceThread(V)首先會檢查 RawBufferParser 內的 bufferQ2 可否讀取資料，當可以讀取資料時，則會將 bufferQ2 內的資料存入暫存器 buffer1，接著將 buffer1 內的影像資料送入 BasicFilterModule 物件(此物件之建置請參考 2.3.3 節)，此物件是負責對媒體資料作壓縮的處理，細節在 3.4.3 會說明。當影像資料自 BasicFilterModule 物件輸出時，為壓縮完成的資料並且切割為預設封包的長度大小，接著會依序存入 buffer1‘。接著呼叫 process()函式，會詢問 BasicMuxModule 內的 bufferQ3 可否放入資料，若可以放入資料，則將 buffer1‘內的封包放入 bufferQ3，並且通知 RawBufferStreamThread(V)隨時可將 bufferQ3 內的封包做發送處理。

SourceThread(V)將封包放入 bufferQ3 之後，會檢查 buffer1’ 內紀錄媒體資料結尾的旗標(FLAG_EOM)，若旗標為 1，表示 buffer1‘內的資料為媒體資料結尾，因此 drainSync 物件會鎖住 SourceThread(V)，此 thread 會被封鎖 3 秒。除非當 RawBufferStreamThread(V)將 bufferQ3 內所存的所有資料送出至網際網路，則 RawBufferStreamThread(V)會執行 drainSync.notifyAll()函式，提早解除 SourceThread(V)的封鎖狀態，使得 SourceThread(V)可以對下一張照

片做壓縮的處理。以上流程如圖 3-16。

當 SourceThread(V)無法讀取 RawBufferParser 內 bufferQ2 的資料時，呼叫 bufferQ2.wait( )函式，bufferQ2 會封鎖 SourceThread(V)，此 thread 會進入封鎖狀態(如圖 3-16 的左圖)，直到 VFWTransferDataThread(V)將資料放入 bufferQ2，接著呼叫 bufferQ2. notifyAll( )函式，通知 SourceThread(V)隨時可以自 bufferQ2 取得資料。

當 SourceThread(V)欲將資料放入 BasicMuxModule 的 bufferQ3 時，若無法將資料放入 bufferQ3，則表示多工器內的 bufferQ3 已存滿資料，因此，呼叫 bufferQ3.wait()函式，bufferQ3 會鎖住 SourceThread(V)，此 thread 會進入封鎖狀態(如圖 3-16 之右圖)；當 RawBufferStreamThread(V)將 bufferQ3 內的資料讀出時，會執行 bufferQ3. notifyAll()函式，通知 SourceThread(V)隨時可以將封包放入 bufferQ3。

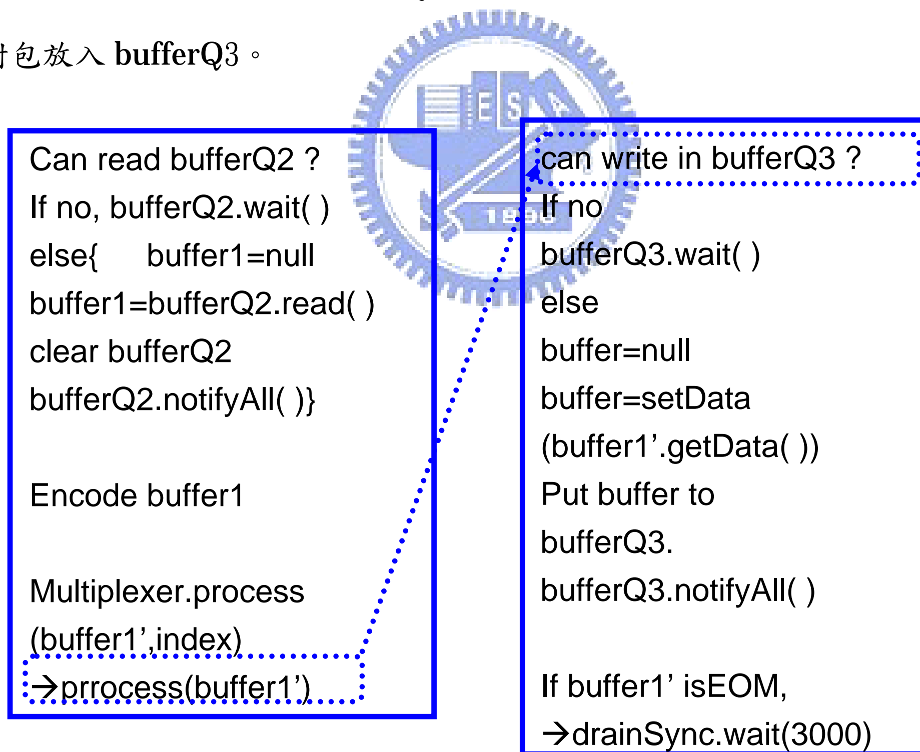


圖 3-16 資料壓縮流程

(4) RTP 封包建立與發送之前置處理

當 RawBufferStreamThread(V)欲將 BasicMuxModule 內 bufferQ3 的封包取出時，會先檢查 bufferQ3 可否讀出資料，當可以讀出資料時，會呼叫

transferData( )函式，接著會再檢查一次 bufferQ3 可否讀出資料，若為可讀，則會將資料存入 RTPSinkStream 物件(此物件之建置請參考 2.3.4)內 buffer2，接著將 buffer2 內的封包建立 RTP 封包，由下層 UDP 送出，以上流程如圖 3-17。

當 RawBufferStreamThread(V)檢查 bufferQ3 可否讀出資料，當不可讀時，會執行 bufferQ3.wait(250)，RawBufferStreamThread(V)會在 0.25 秒後醒來，或是直到 SourceThread(V) 將資料放入 bufferQ3 時，通知 RawBufferStreamThread(V) 隨時可以執行發送封包的處理。當 RawBufferStreamThread(V)呼叫 transferData( )函式時，會再檢查是否可以讀取 bufferQ3 物件內的資料，若此時發現無法讀取，代表之前將資料放入多工器時發生了錯誤，因此 RawBufferStream Thread(V)會進入封鎖狀態，等待資料再度放入 bufferQ3。

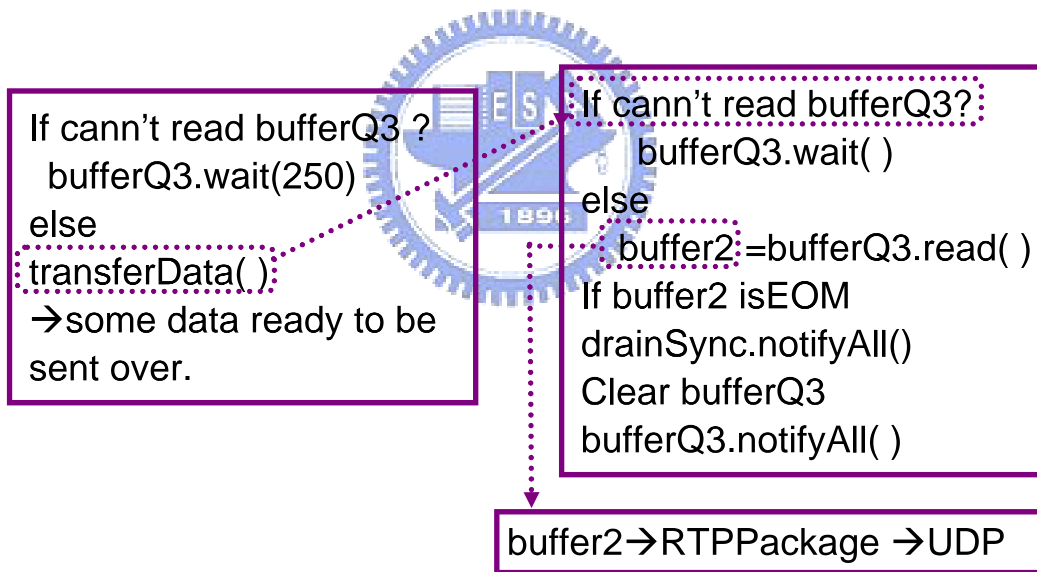


圖 3-17 RTP 封包建立與發送之前置處理

3.4.2 Audio資料的處理流程

圖 3-18 為 Audio 資料的處理流程，在此小節只說明擷取音訊資料，並將音訊資料存入 Parser 的暫存器，後續資料壓縮和封包傳送的流程，與處理影像資料時相同，因此後續部份在此不再說明。

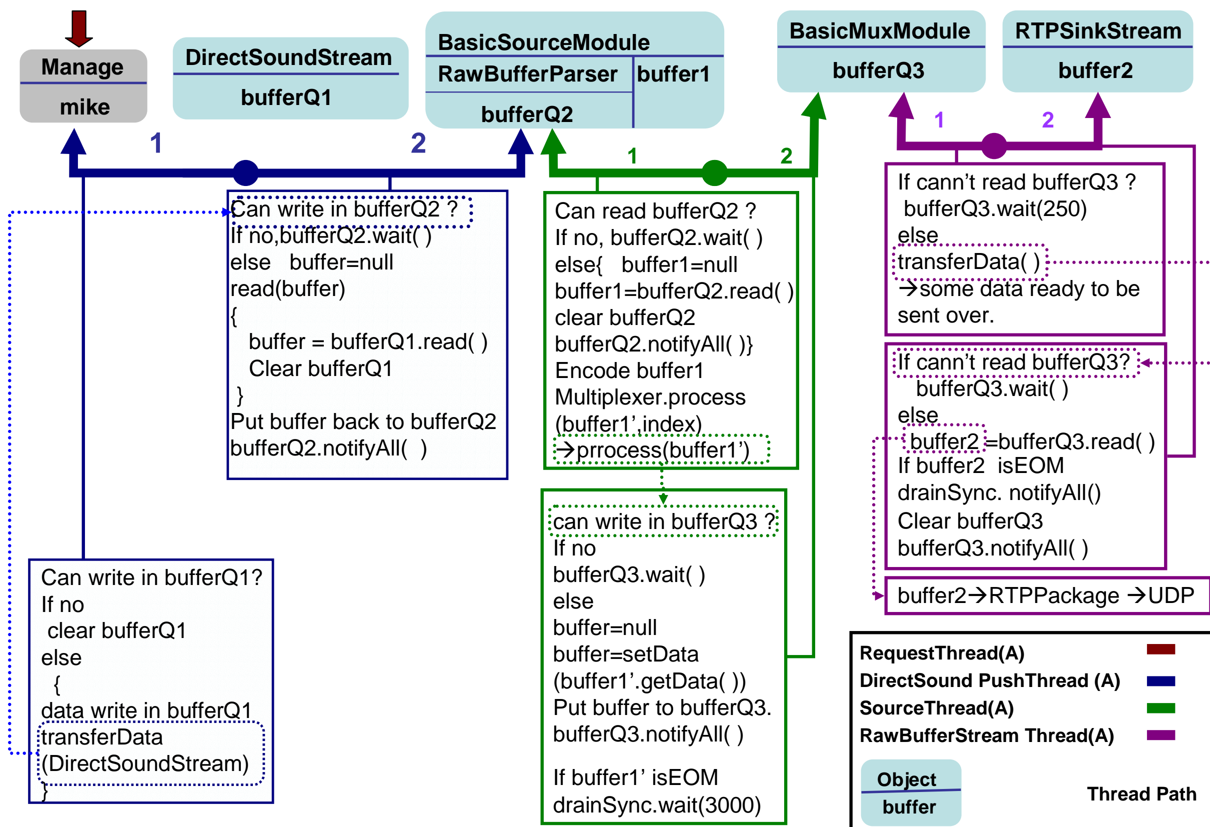


圖 3-18 系統運作(A)

當處理音訊資料時，我們先假設 RequestThread(A)已開啟 mike，接著 DirectSoundPushThread(A)將擷取到的資料存入 bufferQ1，因此會檢查 bufferQ1 可否放入資料，當可放入資料時，會將抓取的音訊資料放入 bufferQ1。接著呼叫 transferData( )函式，檢查 RawBufferParser 內的 bufferQ2 可否放入資料，若可以放入資料時，則會宣告一暫存器 buffer，將 bufferQ1 的內的音訊資料放入 buffer，接著清空 bufferQ1，最後將 buffer 內的資料存入 bufferQ2，以上流程如圖 3-19。當圖 3-19 執行完成時，則表示已將擷取到的媒體資料:音訊資料個別取出，並放入 RawBufferParser 物件，準備進行下一階段的處理。

DirectSoundPushThread(A) 只會在資料無法放入 bufferQ2 時，呼叫 bufferQ2. wait( )函式，此 thread 被 bufferQ2 封鎖，其表示 bufferQ2 不允許放入資料，必須等待 SourceThread(A)讀出 bufferQ2 內的資料時，執行 bufferQ2. notifyAll( )函式通知 DirectSound PushThread(A)隨時可將資料放入 bufferQ2。以上情形如圖 3-19 的右圖。

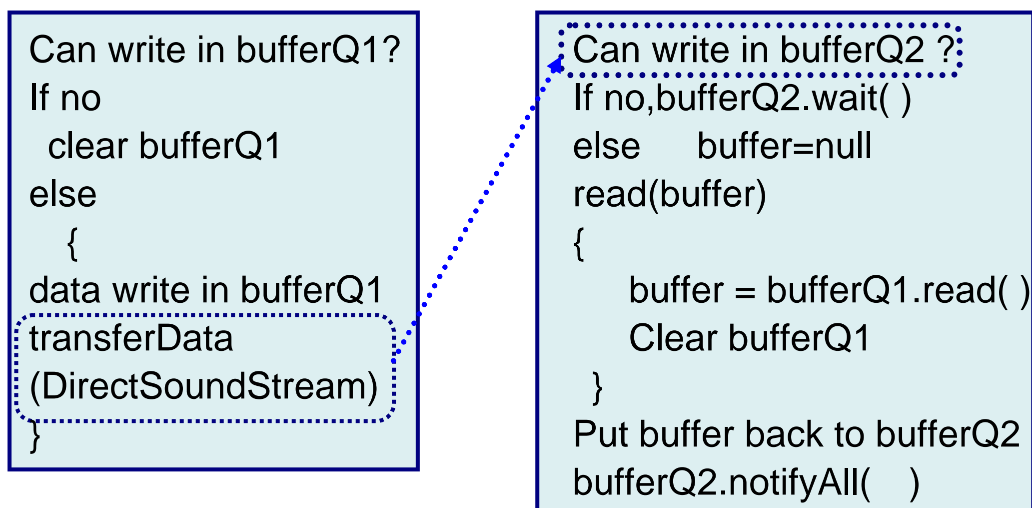


圖 3-19 擷取&抽取音訊資料

3.4.3資料壓縮與傳遞

此小節將說明 SourceThread 執行媒體資料壓縮處理後，資料傳遞至多工器的流程，流程如圖 3-20。在圖 3-20 中，紅色的虛線箭頭表被呼叫的函數執行結束後會再回到 do-while 迴圈，分別為圖 3-20 中 NO. 3 和 NO. 5。SourceThread 將媒體資料自 RawBufferParser 物件讀出並存入 buffer1 物件，接著執行 oc.writeReport( )函式，此函式會將 buffer1 物件內的資料放入 BasicInputConnector 物件內的暫存器，並且呼叫 BasicFilterModule. Process( ) 函式，do-while 迴圈為此函式的主體，以上流程如圖 3-20 中的 NO. 1。

接著執行 ic.getValidBuffer( )函式，將 BasicInputConnector 物件內的媒體資料放入 inputBuffer 物件，接著設置一 outputBuffer 物件，接著執行 codec.process(,)函式，此函式會呼叫 encoder 對 inputBuffer 物件內的資料以數個 macroblock 做壓縮處理，接著將壓縮後的資料存入 outputBuffer 物件。最後對 outputBuffer 物件設置 sequenceNumber 和 TimeStamp，接著程式執行會再回到 do-while 迴圈，以上流程如圖 3-20 中的 NO. 2 與 NO. 3。

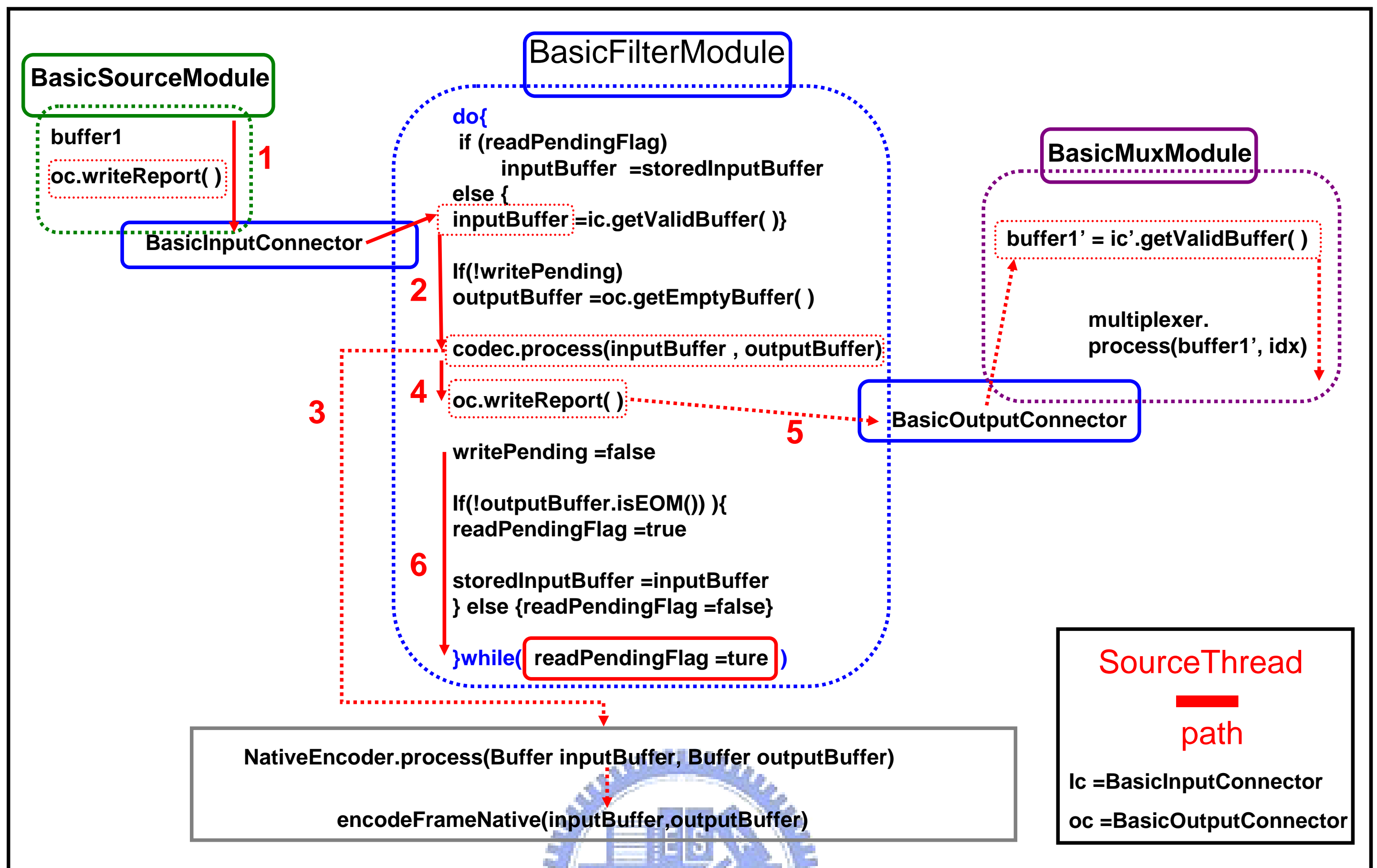


圖 3-20 資料壓縮與傳遞之流程

執行 oc.writeReport()函式，此函式會將 outputBuffer 物件內的資料存入 BasicOutputConnector 物件內的暫存器，接著將暫存器內的資料存入 buffer1'，接著呼叫 multiplexer.process(buffer1', index)將資料放入多工器的 bufferQ3 中，接著程式執行會再回到 do-while 迴圈，以上流程如圖 3-20 中的 NO. 4 與 NO. 5。

設置一邏輯變數:writePending為false，接著執行outputBuffer. isEOM()函式，此函式會檢查outputBuffer物件內FLAG_EOM旗標，當旗標為1時表示此為媒體資料最尾端，並回傳布林值:ture。當判定outputBuffer物件內所存資料為媒體資料最尾端時，會設置一邏輯變數: readPendingFlag =false，當while()判定readPendingFlag =ture條件不成立時，則此迴圈結束執行，此表示inputBuffer物件內的資料已全部壓縮完成並送至多工器，Process()函式也就執行結束，以上流程如圖3-20中的NO. 6。

當判定outputBuffer物件內所存資料並非媒體資料最尾端時，會設置一邏輯變數: readPendingFlag =ture，當while()判定readPendingFlag =ture條件成立時，則此迴圈會繼續執行，此表示inputBuffer物件內的資料並未處理完畢，會再次呼叫encoder執行nputBuffer物件內資料後續的壓縮處理。

3.5 RTP 封包建立與發送

由3.4.3節可知經壓縮處理完成的資料已是符合資料串流的媒體格式，媒體資料將被封裝程RTP封包，送至網際網路，以下以編碼為H.263的影像格式為例加以說明。

由圖3-13可知，buffer1'物件存放壓縮為H.263格式的資料，buffer1'物件為數個picture layer的GOB(groups of blocks)，而buffer1'物件包含H.263的payload header和sub-bitstream，由國際電信聯盟(ITU)所制定的RFC2190，根據[9]，可得知影像封包架構如圖3-21，因此接著必須設置RTP標頭，以下會說明如何設置RTP的標頭。

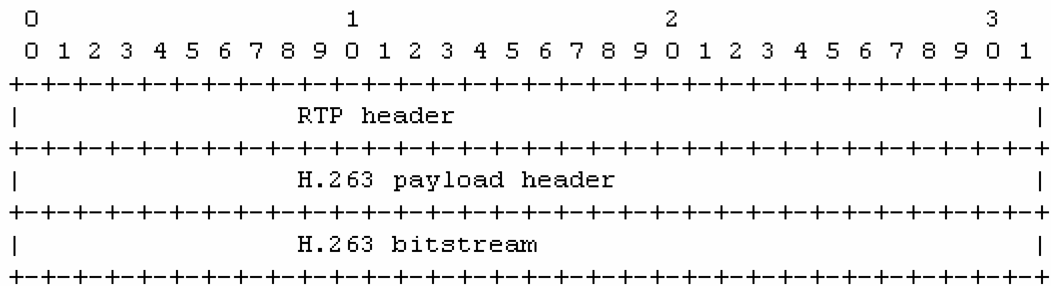


圖3-21 Video Packet Structure

由圖3-13可知buffer2內存放buffer1'的媒體資料，接著檢查buffer2內所存的媒體資料格式是Audio還是Video。若是判斷為Audio則會執行transmitAudio()函式，Video則會呼叫transmitVideo()函式，由於Audio和Video處理的流程相似，因此接下來會以Video的觀點來說明。當呼叫transmitVideo()函式時，會執行以下程序:1.建立RTP封包 2.封包送出至網際網路。

1.建立RTP封包: 將媒體資料(buffer2)封裝為RTP封包，設置RTP標頭所有

參數。由圖2-8，執行RTPTransmitter.transmitPacket(buffer2, SendStream)函式，自buffer2取出TimeStamp的數值，此數值為擷取自系統的時間點，將此數值乘上90除以Time-scale (1000000)，將此數值存入SendStream，此數值為設定RTP 標頭的TimeStamp。

執行MakeRTPPacket(buffer2, SendStream)函式，此函式會建立一RTPPacket物件，其已先預設RTP標頭的V、P、X和CC的參數，接著將buffer2的資料放入RTPPacket物件的data成員。判斷buffer2內標籤(FLAG_EOM)所代表的bit是否為完整frame的結束，若是則會將marker設為1，反之則marker設為0。取出buffer2內的SequenceNumber的數值，將數值填入RTPPacket物件的SequenceNumber成員。最後將SendStream所記錄的TimeStamp、payload type和SSRC的數值填入RTPPacket物件的RTP標頭。

2.封包送出：接著將建立好的RTPPacket物件送至下層的UDP，填入UDP的封包，接著DatagrameSocket將封包送出。由圖2-8，執行RTPRawSender. sendTo(RTPPacket)函式，此函式會以RTPPacket物件、資料長度和用戶端的IP位址為引數建立DatagramePacket物件，此物件將資料位元組填入UDP封包，接著呼叫DatagrameSocket.send(DatagramePacket)將RTP封包送出至網際網路。

3.6 Java Thread

隨著程式設計相關技術的發展與應用面的廣泛，對軟體的要求也越變越複雜，程式的並行性(concurrent)可以說是最典型的例子；基於提高GUI的反應性、增加網路I/O處理量(throughput)…等目的，並行性的程式設計變的相當重要。Java語言本身就有支援並行性程式設計的功能，善用這個功能，就能開發出網路伺服器或高反應性的GUI。

每個電腦程式至少都有一個thread，其為執行應用程式本體的thread。在Java應用程式中，該thread被稱為「main thread」，它以執行類別中main( )函式的第一個述句開始執行的。Java執行緒是由Java虛擬機器(JVM)所產生和管

理，即使只包含一個main函式的Java程式也是以一個單一的thread在JVM下執行。

在Java語言裡，multithread的功能不只是單純的程式庫，而是完全包含在程式語言的規格，其融合了物件導向的特性；物件導向的特性，如動態產生該類別的物件，動態繫結(dynamic binding)，其是指接收到訊息的物件根據該訊息而決定啟動的方式。

Java物件的所扮演的角色是保留給thread處理的定義，其意謂著將物件內所要處理的內容交由thread來處理(圖3-22)。Java物件可以當作多個thread的共用資料，因此，物件同步化的功能就變的相當重要。

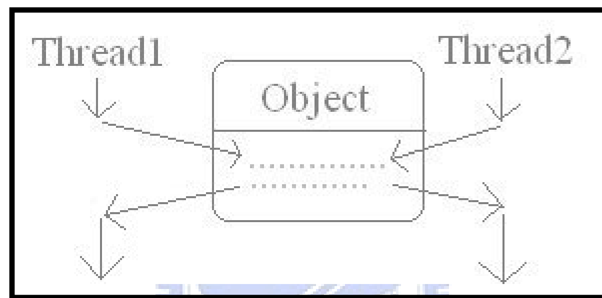


圖 3-22 共用物件

由圖3-23，對於整個Java來說，thread可以利用global memory存取物件內的資料，每個thread還是保有自己的區域變數空間，但物件內的資料是可自動分享。

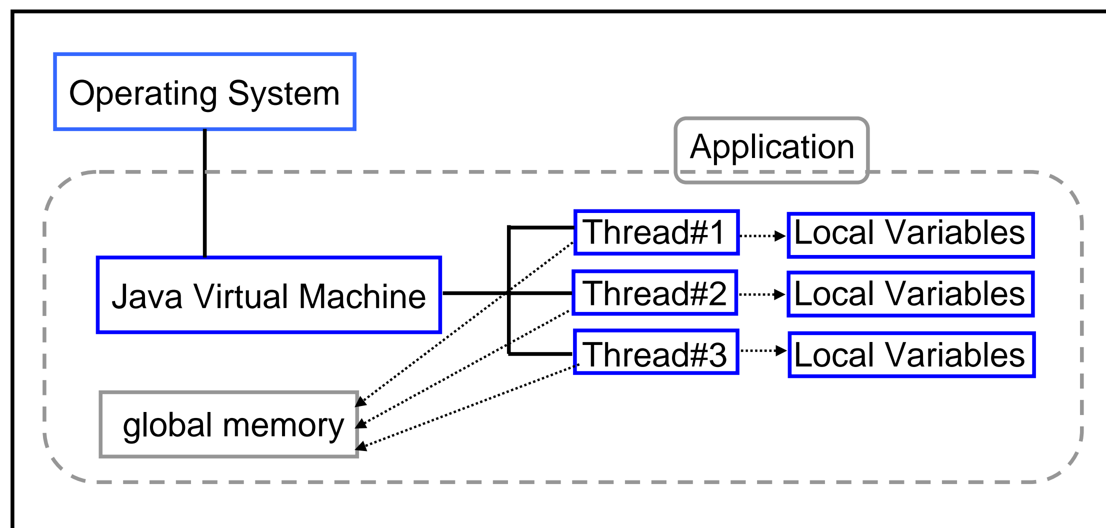


圖3-23 miltithread

3.6.1 Thread狀態

一般來說，當main thread建立新的thread時，此thread會經歷以下幾個狀態(圖3-24)有下列幾種:

1.初始狀態

泛指thread物件已經產生，但尚未執行。

2.待命狀態

當thread呼叫start( )函式，此時thread就會進入待命狀態。

3.執行狀態

當系統分派處理器提供給thread使用時，thread會進入執行狀態。

4.封鎖狀態

當thread有無法繼續往下執行的理由時，或是正在等候回執行狀態，就會進入封鎖狀態。

5.結束狀態

指thread已執行結束的狀態或是拋出一個無法捕捉的例外時，就會進入結束狀態。

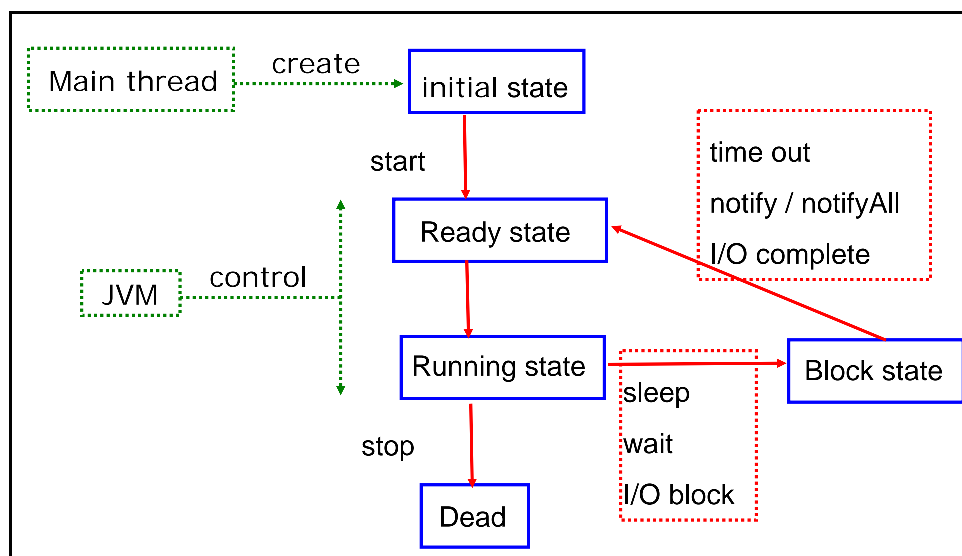


圖 3-24 Thread 物件的狀態圖

Thread必須處於執行狀態才能進行處理，若執行狀態的thread不只一個，則

會由scheduler選擇thread依序執行。當thread物件啟動start( )函式後，新建立的thread即進入待命狀態，當此thread取得執行權，進入執行狀態，接著會執行run( )函式，當run( )函式執行結束時，thread就會進到結束狀態。

若因I/O處理等原因而必須中斷執行動作時，thread會進入封鎖狀態，同理呼叫suspend( )、sleep( )或wait( )函式，會造成thread中斷，而進入封鎖狀態。當I/O處理完成，就會回到待命狀態。因呼叫suspend( )函式，進入封鎖狀態，只要啟動resume( )函式，就會回到待命狀態。若thread因sleep( )函式而進到封鎖狀態，只要等待所預設的時間結束，thread就會回到待命狀態。若thread因wait( )函式而進到封鎖狀態，則呼叫notify( )或notifyAll( )函式，thread即可回到待命狀態。以上關於thread的封鎖控制都是交由Java虛擬機(JMF)管理。

3.6.2 Thread建立的方式

Thread建立的方式有兩種，一種是定義Thread類別的子類別，另一種則是定義Runnable物件。

1.繼承Thread類別

首先要先定義thread類別的子類別。

```
class 類別名稱 extend Thread{
        ......................................
    public void run( ){thread所要處理的程序}
}
```

接著，若要開始執行此thread，則必須產生實體物件，並且呼叫start( )函式。

```
Thread mythread = new 類別名稱 ( );
    Mythread.start( );
```

2. 定義 Runnable 物件

定義介面java.lang.Runnable的實作類別的run( )函式，Runnable介面本身定義如下:

```
public interface Runnable{
    void run( ); }
```

接著必須重新定義run( )函式，並且在run( )函式裡加入thread所要處理的程序。

```
class類別名稱 implements Runnable{
    public void run( ){thread所要處理的程序}
}
```

接著，在thread類別的建構式的參數，設定實作Runnable的物件。

```
target = new 類別名稱( );
Thread mythread = new Thread(target);
mythread.start( );
```

當thread開始執行時，會執行類別名稱內的run( )函式。

3.6.3 Thread同步機制

前面曾提及，Java物件可以當作多個thread的共用資料，意謂著同一個類別可產生多個thread，而程式極有可能接近同時執行此類別內的多個thread，由於這些thread共用相同的成員變數，當修改、刪除或新增成員變數時，很有可能會存取到錯誤的資料。

另一種情況，當兩個thread分別在不同類別時，彼此之間必須要分享資料時，若執行時間也是接近同時，很有可能呼叫同一物件的同一函式，而造成存取資料發生錯誤。

以上都是因為資料不同步而發生的錯誤，為了避免以上的錯誤，Java提供了執行緒同步(thread synchronization)機制，可以將類別中的函式設定為彼此互斥

(mutual exclusion)，可以鎖定存取物件成員的動作，使得在任何時間只有一個物件成員被存取，或是物件的函式只有被一個thread所使用。

Java語言的Thread同步機制必須利用到各個物件的鎖定(lock)，當thread要存取此物件時，必須要先取得一個token，我們稱之為lock。一個物件只有一個lock，當有兩個thread試著要在同一時間存取同一物件，只有一個thread會取得處理此物件的lock，另一個thread必須要等到第一個thread釋放lock才能處理此物件。

當物件處於鎖定的狀態，若有其他的thread想鎖定這個物件，則在物件的lock解除之前，thread會一直處於封鎖狀態。各個物件暗地裡管理，等待並且想要鎖定它們的thread，如圖3-25為thread的集合。

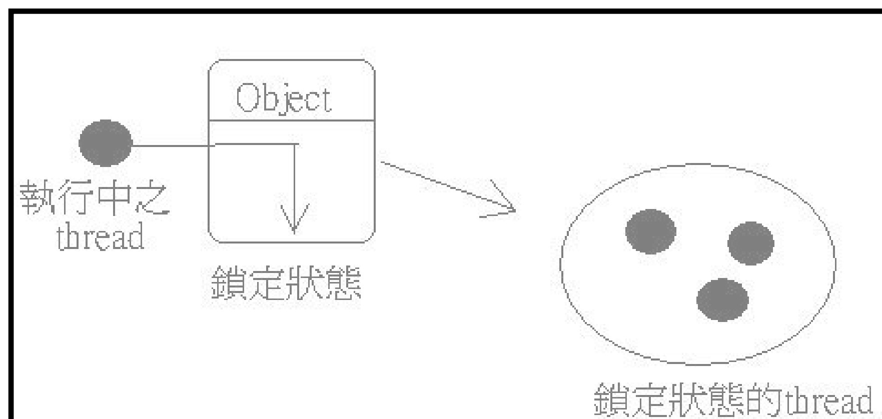


圖3-25 thread的集合

同步機制主要有以下兩種方式:

1.synchronized函式

定義synchronized方法，只須在函式定義前加個修飾子synchronized，如下：

```
synchronized    void working( ){
………………………………
}
```

2. synchronized區塊

另一種以區塊為單位鎖定物件，則可利用整個synchronized區塊。

synchronized區塊是指在大括號裡的程式部份執行時，鎖定小括號裡所設定的物件。例如：

小括號裡設定的是變數Object參照的物件，首先會先鎖定該物件後再執行大括號裡的部份。

```
synchronized(Object){
……………………………
}
```

3.6.4 Thread之間的溝通

由前Thread同步機制，我們以知每個Java物件都有一個lock。每個物件都有提供一個機制能讓它成為等待區；此機制能夠幫助thread間的通訊。此機制背後的想法為：某個thread需要一個特定條件的成立才能執行它的任務，並假設有其它的thread會建立起該條件。當另一個thread建立起該條件時，它會通知前面正在等待該條件的thread。主要是使用下列Object class中的函式來達成:

void wait( )

如果為一thread呼叫wait( )函式時，則此thread會變成封鎖狀態，等待條件的發生，並且解除物件的鎖定，此時處於封鎖狀態的thread則被物件管理。

Void notify( )

若其它thread呼叫notify( )時，會通知正在等待的thread此條件已經發生。但若物件所管理的封鎖狀態的thread有多個，就只會有一個恢復到執行狀態，Java的規格並沒有定義由哪一個thread要收到通知，因此要能夠通知多個thread，該條件已發生，則要呼叫notifyAll( )。

為何要喚醒所有的thread?主要來說，因為無法控制要哪一個thread收到通知，但每一個由通知叫起的thread所等待的條件是不一樣的，因此藉由喚醒所有

的thread，可以將程式設計成由thread自行決定接下來由哪一個thread應該要執行。

Void wait(long timeout)

等待條件的發生。但若沒有在時間內收到通知，thread自然會解除封鎖，再鎖定該物件。

在Java物件都可以呼叫上面的函式，只針對synchronized的物件。

3.6.5 自訂排程

Java對於thread的排程，只依循一簡單的規則。scheduler會從可執行狀態的thread當中，選出順位最高的thread。但並不保證最高順位的thread可以立即執行。但若欲設計成程式不依循排程方針，或想採用特定的排程方針時，必須另外準備自行排程的程式碼。

應用程式階段對於排程的方式有兩種:

1.各thread用wait()保留執行權。

2.由自訂排程管理所有thread的執行權。

第一種是在處理程式碼的前後位置新增同步的程式碼。如下:

```
while(無執行權的條件){
    wait();
}
處理一些程序的程式碼
notifyAll()
```

以上方式可以使應用程式設計更有彈性，而且就算與標準排程同時使用也不會有問題，但也因此要新增同步的程式碼。

第二種很難與標準排程同時使用，因此不加以說明。其他的排程方式有 Round Robin、Time Sharing等。根據[6]、[7]。

3.6.6 Multithread之應用

由3.6.3~3.6.5節已介紹過Thread同步機制、Thread之間的溝通所需使用的函式，以及自訂排程機制。將以上的基本原理做為控制多媒體系統內資料的傳遞與流動，以下我們以兩個thread對暫存器:bufferQ執行存取的程序做為說明，如圖3-26。

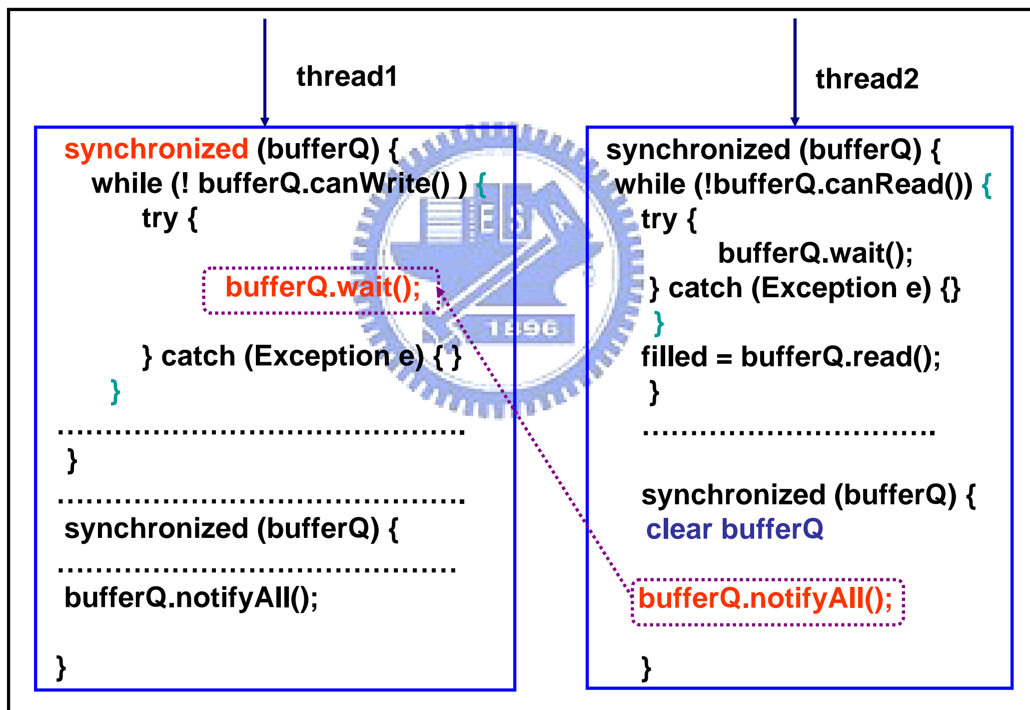


圖3-26thread間之溝通

當bufferQ被宣告為synchronized，thread1要對bufferQ執行放入資料時，thread1必須透過synchronized自JVM取得一個token，可以將它稱之為lock，意謂著thread1鎖住bufferQ，此時若有其它thread想要對bufferQ執行資料存取就只能等待，直到thread1釋放掉lock才能對bufferQ執行資料存取。

當whlie迴圈的條件式成立時，表示thread1不可以對bufferQ放入資料，

此時會執行 bufferQ. wait()函式，thread1 會釋放 bufferQ 的 token，thread1 會變為封鎖狀態，並且解除對 bufferQ 的鎖定，此時處於封鎖狀態的 thread1 則會被 bufferQ 管理。

以相同的同步機制，thread2 要對 bufferQ 執行資料讀出並且放入 filled 暫存器中，接著會將記錄 bufferQ 物件資料量的參數設為 0(稱為清空)，最後會呼叫 bufferQ.notifyAll()函式，藉此函式解除 bufferQ 對 thread1 的管理，thread1 才能繼續執行其它程序。由上可知「thread2 清空 bufferQ」就為 thread1 是可否能對 bufferQ 執行資料放入的條件。

3.6.7 Java虛擬機的簡介

Java虛擬機(JVM)提供multithread的功能，同時使用監視器(monitor)處理同步的問題。其中最特別的thread為垃圾收集thread，其優先權最低，當有thread不再需要執行時或是記憶體空間不足時，垃圾收集thread就自動啟動，以整合零碎自由的記憶空間（fragment），以形成一個較大的記憶區塊，為往後系統需要時做準備。

Java虛擬機定義了不同thread優先權，優先權高表示它將獲得多一些CPU時間；低優先權thread只有當所有高優先權thread被終止後才能取得CPU控制時間。但並不保證高優先權的thread一定會優先處理。

對於任何Java thread，Java虛擬機都必須同時支援：物件的鎖定和thread等待、傳遞通告。物件的鎖定保證thread共享資料時不會互相影響，thread等待、傳遞通告能幫助thread合作互相處理一些共同物件。

Java虛擬機規格明確記載thread必須執行以下兩個動作：

1.從主記憶體複製變數值到它的工作記憶體。

2.從它的工作記憶體寫入值到主記憶體。

Java的監視器支援thread兩種功能:不會互相影響與合作

不會互相影響，是經由在Java虛擬機獨立地將物件鎖定，才能夠使多個thread互相分享資料。

合作，則是經由Java虛擬機使得thread進入封鎖狀態，以及等待類別物件所傳遞的通告，賦予thread相互合作的能力，例如從一個緩衝區讀取資料, 而有另外thread執行寫入資料。以上根據[8]。

第四章 Thread 試驗

4.1 Thread封鎖與時間之試驗

在 3.3.2.1 節「處理影像資料」，相信讀者都會發現到，為什麼要刻意呼叫 sleep( )或是 wait( )函式，並且設置時間限制，使得呼叫這些函式的 thread 能在一定的時間內進入封鎖狀態，或是為什麼一定要設置為此秒數呢?難道設置為其它秒數就不行嗎?以下，本研究會做一些試驗，來說明麼要做如此的設置。

(1) PustThread(V)

PushThread (V)進入執行狀態後，會呼叫sleep(10)函式，封鎖此thread0.01秒，當它再度進入執行狀態時，會呼叫yield( )，以達到和其它thread公平的排程，以避免新的影像資料覆蓋到舊的影像資料。意謂著隔0.01秒後，此thread有機會取得執行權，並且向camera存取影像資料。本實驗重新設置新的秒數(豪秒)如下表格4-1:

表格4-1 thread Sleep試驗

豪秒	0	1	5	10	15	20	25	30	40	50	60	70
CPU%	74	13	16	14	10	15	7	11	8	11	11	22

由表格4-1的數據，可得知休眠期間設為0秒時，很明顯地CPU的負荷比其它秒數據高了許多，換句話說若使此thread修眠的時間期間太短會加重CPU的負荷，如圖4-1和圖4-2。但人類的眼睛對於動態視覺的連續性為每秒30~20張照片，約為33ms~50ms，因此並沒有必要將此試驗中的秒數設置為0，如此做只會加重CPU沒必要的負荷。並且若抓取照片的時間點相距太近則會抓取幾乎相同的照片，而負責儲存擷取資料的暫存器也可能因空間不足而丟棄之前所儲存的資料，因此必須設置適當的休眠時間，由此可知設置sleep()函式對整體系統而言是必要的。

在此試驗中，使得此thread修眠的時間期間設為50ms秒以後，會發現圖片會非常不連續，因此不可以將秒數設為50ms以後。由上可得適當的知休眠期間為0~50(豪秒)。但經由觀察可發現設置休眠時間約在10ms時，播放影像的流暢度是最好的。

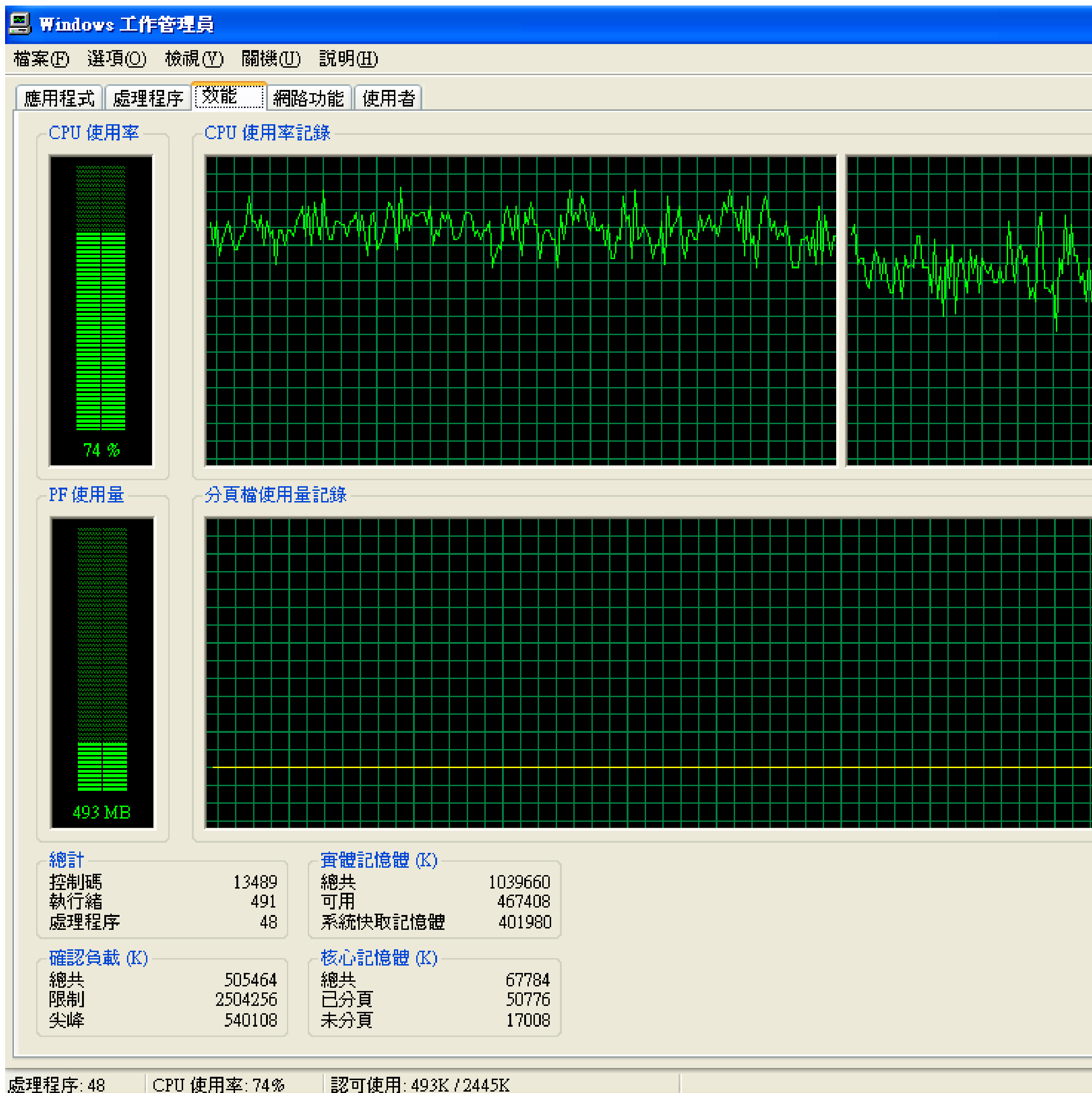


圖 4-1 sleep(0)，CPU之使用率

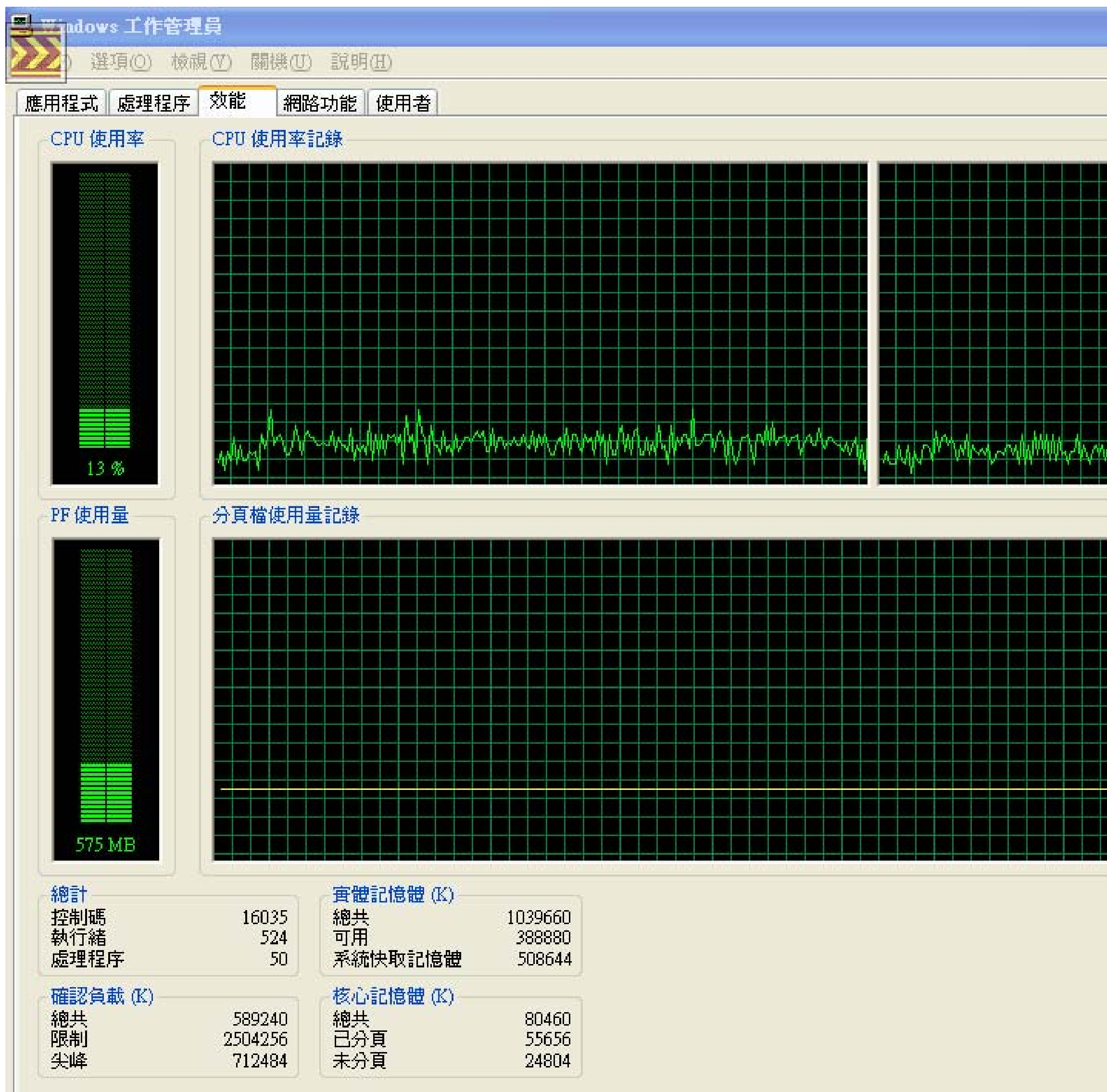


圖 4-2 sleep(10)，CPU 之使用率

(2)VFWTransferDatathread(V)

原先的預設為VFWTransferDatathread(V).wait(250)，當無資料可以讀出時，此thread會進入封鎖狀態0.25秒，本實驗重新設置新的秒數(豪秒)如下：0、5、10、70、800，對於CPU的負荷或是影像播放都是沒有任何影響，因為此thread的可執行條件成立時，會收到解除封鎖的通告，此與時間是沒有任何關係的，但希望此thread能在0.25內能重新回到待命狀態，或許當它在排隊取得執行權時已有媒體資料進入暫存器，就不必再等待解除封鎖的通告。

RawBufferStreamThread(A/V)與VFWTransferDatathread(V)的原理是相

同的，因此試驗的結果也會相同。其中wait( )與wait(0)函式所執行的功能是相同的。

4.2 Thread 執行次序和時間之量測

此小節，為探討本系統處理影像資料相關 thread 執行的順序，當一 thread 開始執行資料存取的程式碼時，在此試驗我們將會抓取此 thread 開始執行的時間點。在稍後的說明中，thread(0)表示為 VFWRequestThread(V)；thread(1)表示為 PushThread(V)；thread(2)表示為 VFWTransferDataThread(V)；thread(3) 表示為 SourceThread(V) ；thread(4) 表示為 RawBufferStreamThread(V)。

由 3.3.2.1 節，可得知每一個 thread 要對暫存器執行資料存取的程式碼前，都必須要檢查暫存器的狀態，由圖 4-3 中，bQ1、bQ2 和 bQ3(暫存器)都有各自專屬的 thread 執行資料傳遞，當檢查暫存器的狀態為無法執行時，thread 就會被它所檢查的暫存器鎖住，而無法執行。

由於在此是取得「thread 開始執行資料存取的程式碼」，其代表所取得為 thread 開始執行的時間點，若為 thread(1)表示已睡醒且執行過 yield( )函式，或為 thread(2)、(3)、(4)表示已通過第一個暫存器的狀態檢查，並且已開始執行資料存取的程式碼，但由圖 4-3 中，thread(1)、(2)和(3)必須檢查第 2 個暫存器的狀態，有可能被暫存器管理，而進入封鎖狀態，在圖 4-4 中我們會驗證第三章的理論。

4.2.1 暫存器之儲存列

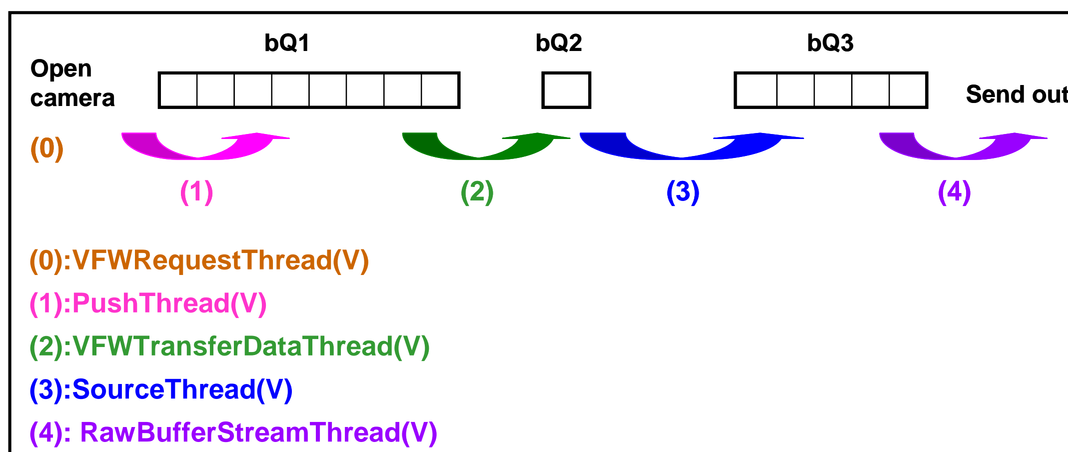
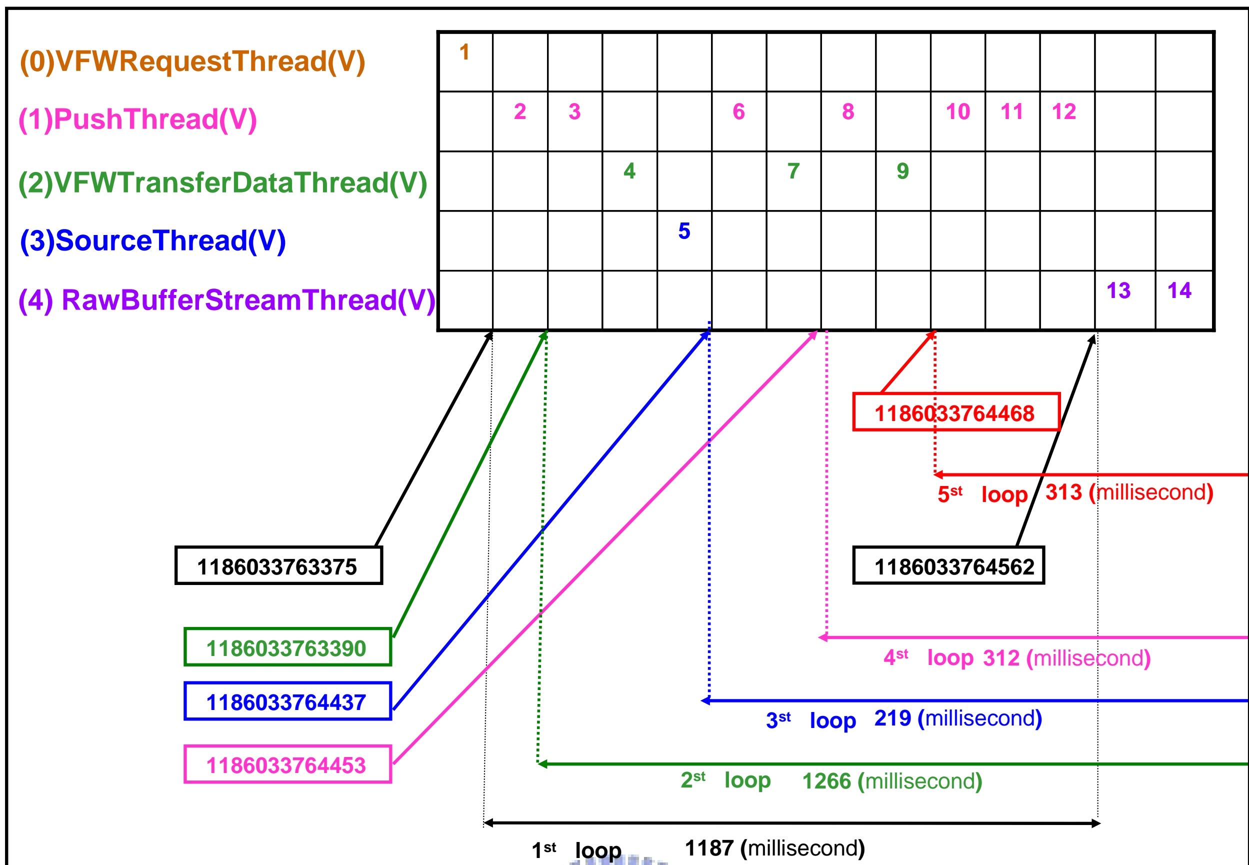


圖 4-3 暫存器內容

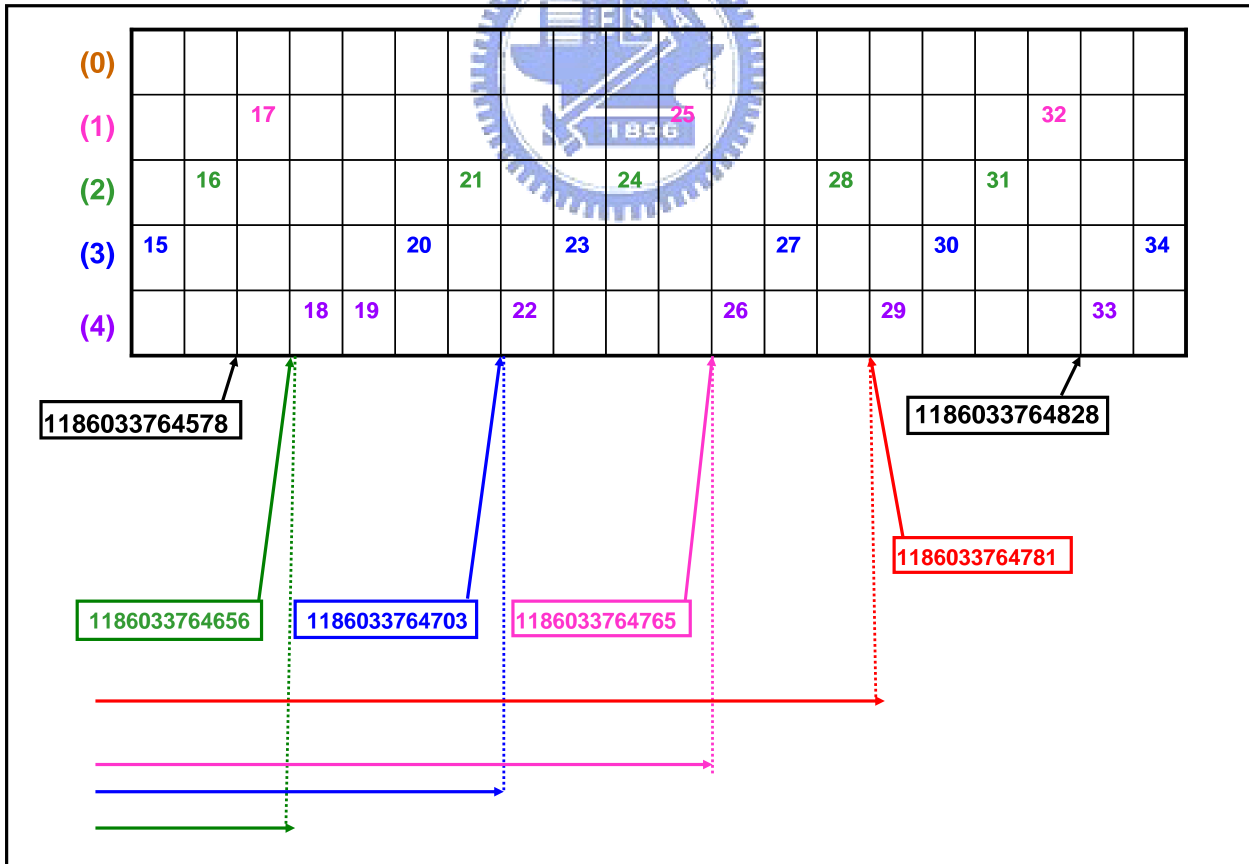
由圖 4-3 中，bQ1 的每一個 queue 為一張照片的資料量，bQ2 也為一張照片的資料量。thread(3)會將 bQ2 內的資料做壓縮並且切割為符合預設 RTP Payload 大小(小於 984 byte)資料區段，並將資料區段逐一放入 bQ3 內的 queue，最後由 thread(4)將 queue 內的資料加上 RTP 標頭，建立為 RTP 封包接著自 Socket 發送。

4.2.2 次序和時間量測

在此小節稍後的說明中，thread(0)表示為 VFWRequestThread(V)；thread(1) 表示為 PushThread(V)；thread(2) 表示為 VFWTransferDataThread(V)；thread(3)表示為 SourceThread(V)；thread(4) 表示為 RawBufferStreamThread(V)。



(a)



(b)

圖 4-4 順序和時間量測(a)(b)

圖 4-4 中，縱軸表格中的號碼表示為 thread(0)~(4)的執行順序，橫軸表示此 thread 的起始時間點，在此只標出部分觀察點，實際的部分數據請參考附錄一。

此小節主要分為四部分：(1)Thread 執行次序，說明 thread 能依據 3.3.2 節的設計概念依序執行；(2)處理媒體資料的時間；(3)照片傳送時間點與 bQ3 不足時的應變機制；(4)鎖住 thread3 的時間試驗。

(1)Thread 執行次序

由 4.2 節可以得知處理影像資料的 thread，進入待命狀態的順序為 thread(1)、(2)、(3)、(4)。由圖 5-4(a)中，NO. 2、3 為 thread(1)執行兩次，表示此時還未有其它 thread 進入待命狀態。接下來若 thread (2)、(3)進入待命狀態，thread (1)則必須讓出執行權，thread (3)若取得執行權就會被 bQ2 鎖住，因此，此時能執行的唯有 thread(2)，thread (3)必須要等 thread (2)執行完後才能執行。請參考表格 4-2 中 NO. 2~4。

NO. 5 為 thread(3)執行，由 4.4 節可知 thread(3)完成執行資料寫入 bQ3 時，將被 drainSync 物件鎖住，或 bQ3 物件不允許寫入而被 bQ3 物件鎖住，除非 thread(4)有起來執行，否則 thread(3)不可能解除封鎖。NO.9 為 thread(2)自 bQ1 取出資料欲存入 bQ2 中，由表格 4-2 可知 bQ2 不允許寫入資料，因此 thread(2)被 bQ2 物件鎖住，除非 thread(3)有執行否則無法解除封鎖。請參考表格 4-2 中 NO. 5~9。

由 NO. 5 和 9，可知 thread(2)和 thread(3)已進入封鎖狀態，此時若 thread(4)還未進入待命狀態，就唯有 thread(1)可執行。當 thread(4)進入待命狀態時，thread(1)若欲執行會因 yield()函式而放棄執行權再次進入待命狀態，thread(2)和 thread(3)已知為封鎖狀態，因此，NO. 9 為 thread(4)執行的開始時間點，thread(4)接著會解開 drainSync 物件或 bQ3 物件對 thread(3)的鎖定。請參考表格 4-2 中 NO. 10~14。

NO.15 也唯有 thread(3)能執行，因 thread(2)在封鎖狀態並且 thread(1)若欲執行會因 yield()函式而放棄執行權再次進入待命狀態，只有 thread(3)能執行，並且解開 bQ2 物件對 thread(2)的鎖定。

由前的討論可知，thread 會依據 3.3.2 節的設計概念依序執行，可以自表格 4-2 和圖 4-4 看出 thread 有一定的規律。

表格 4-2 暫存器內容

執行順序 (NO.)	bQ1 (數量)	bQ2 (數量)	說明
2	1	0	
3	2	0	
4	1	1	
5	1	0	thread (3)被 drainSync 物件或 bQ3 物件鎖住(在此例為前者)
6	2	0	
7	1	1	
8	2	1	
9	2	1	此時 bQ2 以不允許放入資料，thread(2)被 bQ2 物件鎖住。
10	3	1	
11	4	1	
12	5	1	
13	5	1	解開 drainSync 物件或 bQ3 物件對 thread(3)的鎖定(在此例為前者)
14	5	1	
15	5	0	解開 bQ2 物件對 thread(2)的鎖定
16	4	1	

(2)處理媒體資料的時間

由圖 4-4 可觀察出處理每一張照片所必須花費的時間，由表格 4-3 中，每一

個 Loop 表示一張照片自抓取、壓縮至封裝 RTP 封包並開始自 Socket 發送所花費的時間。表格 4-3 中為可以發現前 2 次所花費的時間特別多，這是因為 main thread 此時還未執行完成，直到 main thread 呼叫 thread(4)的 start()函式之前，thread(1)、(2)和(3)已經先在待命狀態，自然隨時可以先行取得執行權。

我們也可以發現每一個 Loop 彼此間所花費的時間是有重複的，以圖 4-4 中 NO. 2~13 為例，NO. 6 為 thread(3)執行，當 thread(3)執行結束理論上要執行 thread(4)，但由於前一段說明的理由，以及 Java thread 同步機制的特性，勢必會為下一次要處理的資料先做處理。經由統計 13 張照片處理的時間(表格 4-3)，可以發現後期的第 3~13 張所花費的時間是比較少的，其平均值約為 241.8 (millisecond)，因此也可以得知每個 thread 彼此間切換的時間花費是很少的。

因此可以得知要依序以 thread(1)、(2)、(3)、(4)執行，基於上面敘述的理由，在 multithread 是很難做到的，但也因 thread 彼此的牽制使得 thread 之間能相互合作。

表格 4-3 處理資料時間表

Loop	1	2	3	4	5	6	7
Time(millisecond)	1187	1126	219	313	313	344	328
Loop	8	9	10	11	12	13	
Time(millisecond)	297	203	141	157	157	188	
Average(3~13)	241.8 (millisecond)						

(3)照片傳送時間點和**bQ**3不足時的應變機制

由3.3.2節可知當thread(3)執行時，最後一定會將媒體資料的尾端放入bQ3，而此時會執行drainSync.wait(3000)函式，thread(3)會被drainSync物件鎖住3秒，3秒後會自動解開封鎖。除非thread(4)將bQ3中的資料送到Socket發送至網際網路，當thread(4)送出的封包為媒體資料的尾端就會執行drainSync.notifyAll( )函式，提早解開drainSync物件對thread(3)的封鎖。由圖4-4中，NO. 5

和NO. 15之間差了328(millisecond)並未超過3秒，因此我們可以判定，NO. 13和NO. 14必定將一張照片的所有封包發送自網際網路。

但thread(4)執行封包發送時，不一定會連續將bQ3內的封包送完，也可能分為數次，實際數據請參考附件二，紫色方框內可以觀察到thread(4)將bQ3內最後一個封包送出之前有發生其它thread起來執行的情況。會發生此情形是因為網路發生壅塞的情形或是作業系統將執行權交給其它thread。

在此我們並不知道thread(3)壓縮後，會產生多少資料區塊逐一放入bQ3中，因此bQ3很有可能會有空間不夠的問題，由附件二我們觀察到thread(4)執行了6次，其表示當thread(3)無法將壓縮後的資料放入bQ3時，bQ3會鎖住thread(3)，直到thread(4)將資料自bQ3取出時，就會解除bQ3對thread(3)的封鎖，thread(3)則會把壓縮完成的資料放入bQ3，完成一張照片的壓縮，thread(4)會再執行送出bQ3內的資料至Socket，因此附件二中會有顯示thread(4)執行了6次。

(4)鎖住thread3的時間試驗

由「(3)照片傳送時間點和bQ3不足時的應變機制」可知利用drainSync. wait(3000)函式可以保證thread4必定會將bQ3內所有關於一張照片的封包全部送出，thread3才會壓縮下一張照片，由此才能達到即時送出一張照片的封包，不會造成bQ3內有兩張照片的封包使得即時傳送的效益變差。

由附件三之(1)~(4)可以觀察到當thread3壓縮完一張照片後，並沒有做傳出封包的處理，接續反而是做壓縮第二張照片的處理，很明顯地即時傳送的效益變差，因此設置此機制才能改善即時傳送的效益。由表4-3可知封鎖時間的設置至少要大於1.2秒，才能保證照片壓縮完後，會先做封包傳輸的處理。

(0)	1																			
(1)		2	3			6		8				12			15	16	17		19	20
(2)				4			7		9					14						
(3)					5						11									
(4)										10			13					18		

						27							34										44	
		23			26				30			33				37			40			43		
	22		24					29			32				36			39			42			
21				25			28			31				35			38			41				45

圖4-5 無檢查EOM時的執行順序

圖 4-5 中，縱軸表格中的號碼表示為 thread(0)~(4)的執行順序，橫軸表示此 thread 的起始時間點，在此只標出部分觀察點，實際的部分數據請參考附錄三之(1)。

觀察圖4-4和4-5，可發現兩者在穩態時thread(2)~thread(4)的執行順序大致是相同的，不一樣處是在圖4-5中的NO. 22~24，此處為連續壓縮兩張照片的情形，造成即時傳送的效益便差，此情形是在圖4-4中不曾發生的，因圖4-4為設置drainSync.wait(3000)的機制時，thread的執行狀況。此機制就可以防止圖4-5中的NO. 22~24此情形發生。

4.3 結論與相關探討

(1)Thread的數量

對於本章的第一節，本論文點出在此系統，處理影音資料總共會用到9個thread，處理影像使用5個thread，處理音訊使用4個thread。讀者一定感到困惑，為什麼一定要用到9個thread，一個或是3~4個thread不行嗎?或是，使用更多的thread不也一樣可以嗎?

當只使用單一thread，在所有子系統的暫存器間，往返搬運媒體資料，所有的執行順序都必須依序執行，以此系統為例，自camera抓取媒體資料，放入parser，作資料壓縮，送入網際網路，這一連串的程序是相當冗長的。當程式執行至將封包送入網際網路時，程式必須要回到最初的子系統物件，也就是自camera抓取媒體資料，當程式要回到此程序時，必須要經過多層函式的呼叫，

才能回到此程序，因此就浪費了許多時間在做多層函式的呼叫，整個程式的效率就會相當的差，相對的接收端做播放時就會有很大的延遲，因此本研究不使用單一thread。

本研究是依子系統的各別功能，而由專屬的thread來執行任務，為了就是要解決上述的問題，只要thread所要執行的是多個程序，就會發生上述效率變低的問題。但若所使用的thread過多呢?此種情形，就會浪費作業系統的資源，每一個thread都會佔有一記憶體的位置，也就是當產生一新的thread就會消耗記憶體空間，切換thread時會產生額外的負擔，因此不應該使用過多的thread。

(2)排程機制的選擇

本研究所選用的排程機制是自定排程，並且配合Java虛擬機的標準排程機制，並不使用Round Robin或Time Sharing等排程機制。以Round Robin來說，雖然子系統應該會照著一定的順序執行，但在一些條件下希望能讓給其它thread先執行，例如前面所提及SourceThread會將壓縮完成的資料放入多工器的暫存器中，當發現此媒體資料是結尾時，就會鎖住此thread，讓其它thread先執行。或是當多工器的暫存器以被寫滿時，會由其它thread將資料讀出，再將後緒未完成壓縮的資料寫入多工器。

以 Time Sharing 來說，我們很難預測每一個 thread 要執行多久，很有可能有些 thread 處理媒體資料時，還未處理完成，其執行權就由 Java 虛擬機收回了，因此，此種排程機制並不適合本系統。

(3)優點與缺點

在本章節中，我們可以觀察到 thread(1)~(4)並沒有依照順序執行，完全是依照作業系統、資料壓縮和暫存器當時的實際情況而執行的。在實際的系統中，是無法保證當時作業系統的狀態或是網路的狀態，舉例來說，在網路發生壅塞時，thread(4)暫時無法送出封包，而先交由其它的 thread，因此在等待 thread(4)可以執行前，可以由其它的 thread 執行其它任務，才不會有整個系統停止運作

的情形發生。

但是，以處理一張照片的觀點來看，因 multithread 可能無法依序執行，因此使得處理的時間變長，此是因為處理每一張照片的時間都有重疊到，當處理此次的照片時可能為下一次的處理做準備，因而拉長了此次處理照片的時間。

第五章 結論

由本論文的第二章到第三章中，可以了解到擷取影音資料，直到傳送至遠方用戶端所需的處理程序。其中包含了許多領域的相關知識，如各種影音壓縮知識、網路程式設計和通訊協定。探討藉由JMF API對於處理原始的媒體資料，在每一個階段所提供的功能，以及Java Socket所提供有關於Socket API套件。

整個系統的運作，藉由建立數個 thread 和 thread 的同步機制，我們可以達到對數個 thread 某種程度上的控制，使得 thread 之間相互合作，所有的子系統達到制衡，換句話說就是 thread 之間達到制衡以執行資料的傳遞，使得系統整體比較單一 thread 執行效率較高。

經由第四章的試驗，可以了解到要控制 camera 適當的休眠時間是不太容易的，必須要考慮到 CPU 的使用率和抓取照片的時間點。利用 drainSync.wait(time)的機制可以改善即時傳輸的效益。以暫存器來做為 thread 之間溝通的橋樑可以使得負責儲存封包的暫存器不需要太大，利用 thread 之間的制衡達到暫存器再利用的好處。

由於本論採用為主動傳輸串流資料至接收端，建立RTP封包送至遠方的用戶端，傳輸端與用戶端之間並無連線溝通的程序，因此只達到作為主動傳輸多媒體資料的系統，並未提供伺服器的功能，因此還有很大的發展空間。

若要將本論文所提的系統發展為一串流伺服器，可藉由2.5.1節所介紹的Java Stream Socket API建立伺服端，使用API的相關函式處理用戶端主動連線的要求，再由本論文所提傳輸多媒體串流資料的系統作為OSI中的Transport layer傳送RTP封包至用戶端。

若希望能同時接受數個用戶端的要求，可配合3.6節Java Thread，對於每個用戶端的要求，由一thread來處理，就可以達到伺服端處理多個用戶端連線的要求。以上都是本論文未來可以更進一步去研究的幾個方向。

參考文獻

[1]Java 網路程式設計 作者:黃嘉輝 出版:文魁

[2]http://java.sun.com/j2se/1.5.0/docs/api/

[3]Java 網路程式設計 作者:elliotte Rusty Harold 譯者:蔣大偉 出版: 歐萊禮

[4]H. Schulzrinne, Columbia University, "RTP: A Transport Protocol for Real-Time ",RFC3550. Titl, July 2003.

[5]http://java.sun.com/products/java-media/jmf/index.jsp

[6]Java Thread 深度探討 作者:Soctt Oaks&Henry Wong 譯者:楊尊一

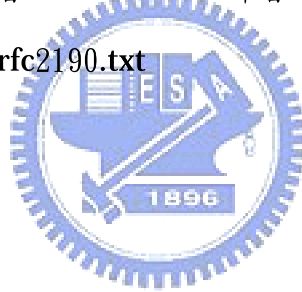
出版: 歐萊禮

[7]Java2 物件導向技術專題 Design Pattern、Framework、Multithread、

Concurrent 作者:戶松豊和 譯者:李于青 出版:博碩

[8]Java 虛擬機深入解析 作者:Bill Venners 譯者:葛湘達 出版:麥格羅.希爾

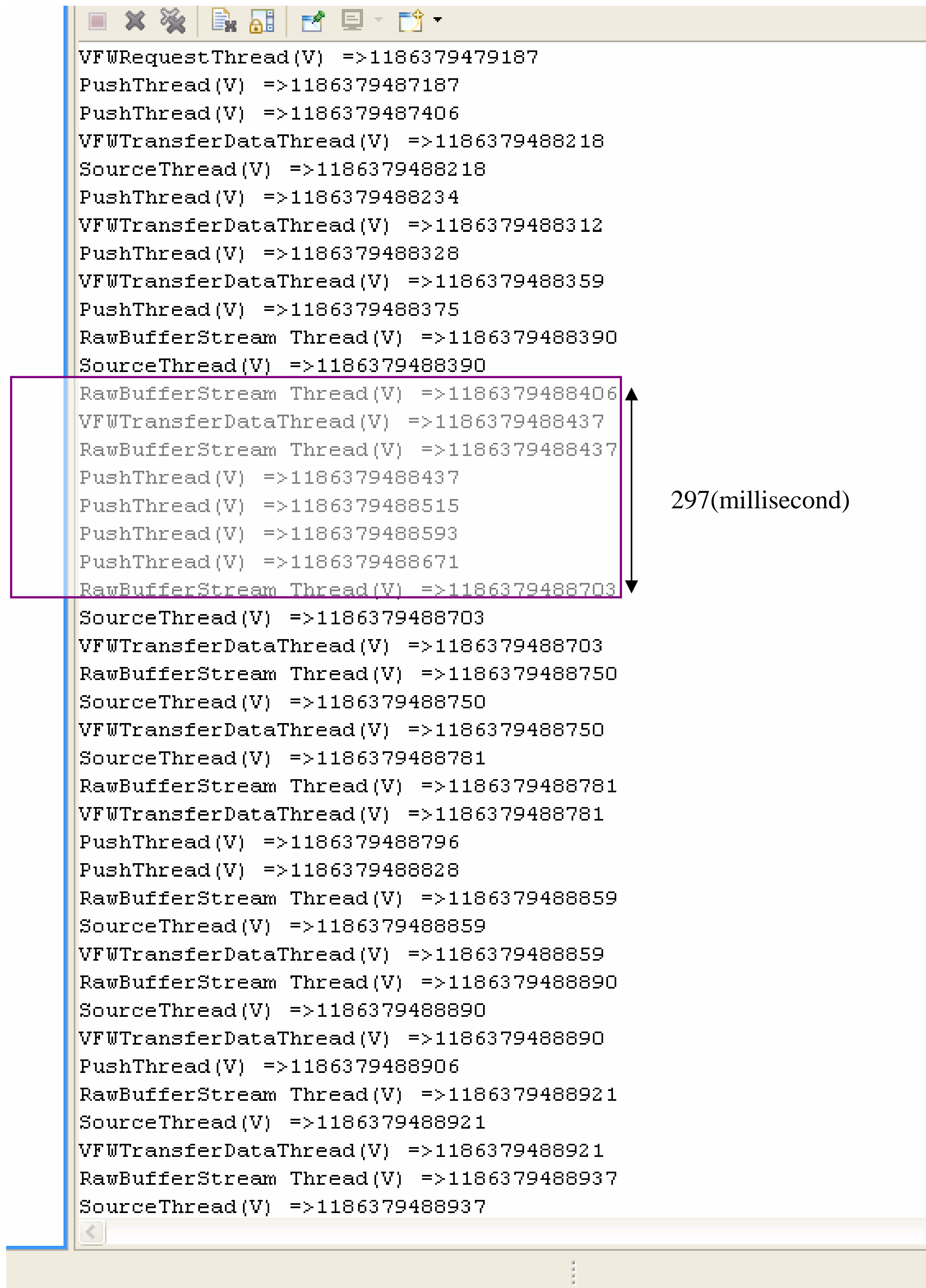
[9]http://www.ietf.org/rfc/rfc2190.txt



附件一 Thread 順序和時間量測的數據

```
VFWRequestThread(V) =>1186033757703
PushThread(V) =>1186033763375
PushThread(V) =>1186033763390
VFWTransferDataThread(V) =>1186033764234
SourceThread(V) =>1186033764234
PushThread(V) =>1186033764437
VFWTransferDataThread(V) =>1186033764437
PushThread(V) =>1186033764453
VFWTransferDataThread(V) =>1186033764453
PushThread(V) =>1186033764468
PushThread(V) =>1186033764484
PushThread(V) =>1186033764531
RawBufferStream Thread(V) =>1186033764562
RawBufferStream Thread(V) =>1186033764578
SourceThread(V) =>1186033764578
VFWTransferDataThread(V) =>1186033764578
PushThread(V) =>1186033764578
RawBufferStream Thread(V) =>1186033764656
RawBufferStream Thread(V) =>1186033764671
SourceThread(V) =>1186033764671
VFWTransferDataThread(V) =>1186033764671
RawBufferStream Thread(V) =>1186033764703
SourceThread(V) =>1186033764703
VFWTransferDataThread(V) =>1186033764703
PushThread(V) =>1186033764718
RawBufferStream Thread(V) =>1186033764765
SourceThread(V) =>1186033764765
VFWTransferDataThread(V) =>1186033764765
RawBufferStream Thread(V) =>1186033764781
SourceThread(V) =>1186033764781
```

附件二 暫存器空間不足的特例



```
PushThread(V) =>1186379489859
RawBufferStream Thread(V) =>1186379489859
VFWTransferDataThread(V) =>1186379489906
SourceThread(V) =>1186379489906
PushThread(V) =>1186379489906
RawBufferStream Thread(V) =>1186379489921
VFWTransferDataThread(V) =>1186379489968
SourceThread(V) =>1186379489968
PushThread(V) =>1186379489984
RawBufferStream Thread(V) =>1186379489984
VFWTransferDataThread(V) =>1186379490031
SourceThread(V) =>1186379490031
RawBufferStream Thread(V) =>1186379490031
PushThread(V) =>1186379490046
VFWTransferDataThread(V) =>1186379490093
SourceThread(V) =>1186379490093
PushThread(V) =>1186379490109
RawBufferStream Thread(V) =>1186379490109
VFWTransferDataThread(V) =>1186379490187
SourceThread(V) =>1186379490187
RawBufferStream Thread(V) =>1186379490203
PushThread(V) =>1186379490203
VFWTransferDataThread(V) =>1186379490265
SourceThread(V) =>1186379490265
PushThread(V) =>1186379490265
RawBufferStream Thread(V) =>1186379490281
VFWTransferDataThread(V) =>1186379490375
SourceThread(V) =>1186379490375
PushThread(V) =>1186379490375
RawBufferStream Thread(V) =>1186379490375
VFWTransferDataThread(V) =>1186379490390
SourceThread(V) =>1186379490390
PushThread(V) =>1186379490406
RawBufferStream Thread(V) =>1186379490406
RawBufferStream Thread(V) =>1186379490421
RawBufferStream Thread(V) =>1186379490421
RawBufferStream Thread(V) =>1186379490421
RawBufferStream Thread(V) =>1186379490421
RawBufferStream Thread(V) =>1186379490421
VFWTransferDataThread(V) =>1186379490437
SourceThread(V) =>1186379490437
PushThread(V) =>1186379490453
```

bQ3 不足

2007年8月6日

附件三 SourceThread(V)封鎖試驗

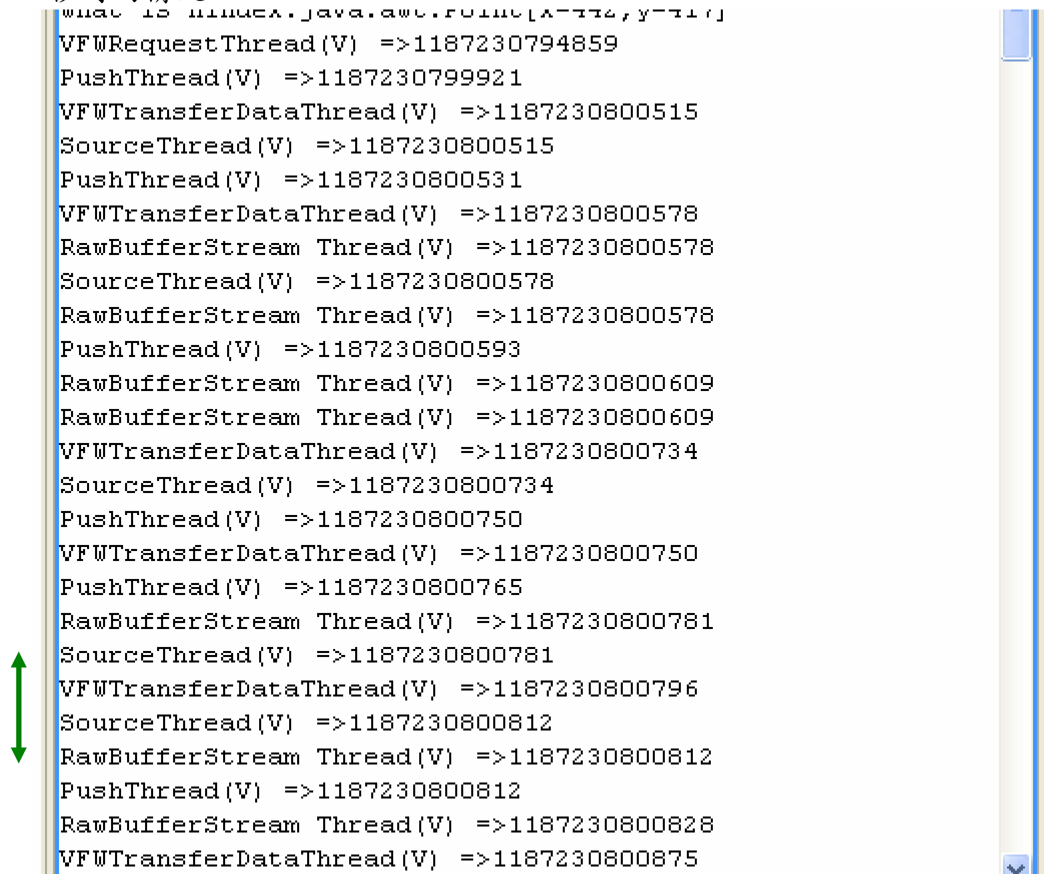
(1)無設置時的情況

```
VFWRequestThread(V) =>1187254980375
PushThread(V) =>1187254988078
PushThread(V) =>1187254988109
VFWTransferDataThread(V) =>1187254988968
SourceThread(V) =>1187254988968
PushThread(V) =>1187254988968
VFWTransferDataThread(V) =>1187254989000
PushThread(V) =>1187254989015
VFWTransferDataThread(V) =>1187254989078
RawBufferStream Thread(V) =>1187254989078
SourceThread(V) =>1187254989078
PushThread(V) =>1187254989078
RawBufferStream Thread(V) =>1187254989453
VFWTransferDataThread(V) =>1187254989453
PushThread(V) =>1187254989453
PushThread(V) =>1187254989468
PushThread(V) =>1187254989484
RawBufferStream Thread(V) =>1187254989484
PushThread(V) =>1187254989500
PushThread(V) =>1187254989500
RawBufferStream Thread(V) =>1187254989515
SourceThread(V) =>1187254989515
VFWTransferDataThread(V) =>1187254989515
SourceThread(V) =>1187254989546
RawBufferStream Thread(V) =>1187254989546
VFWTransferDataThread(V) =>1187254989546
PushThread(V) =>1187254989562
RawBufferStream Thread(V) =>1187254989578
SourceThread(V) =>1187254989578
```

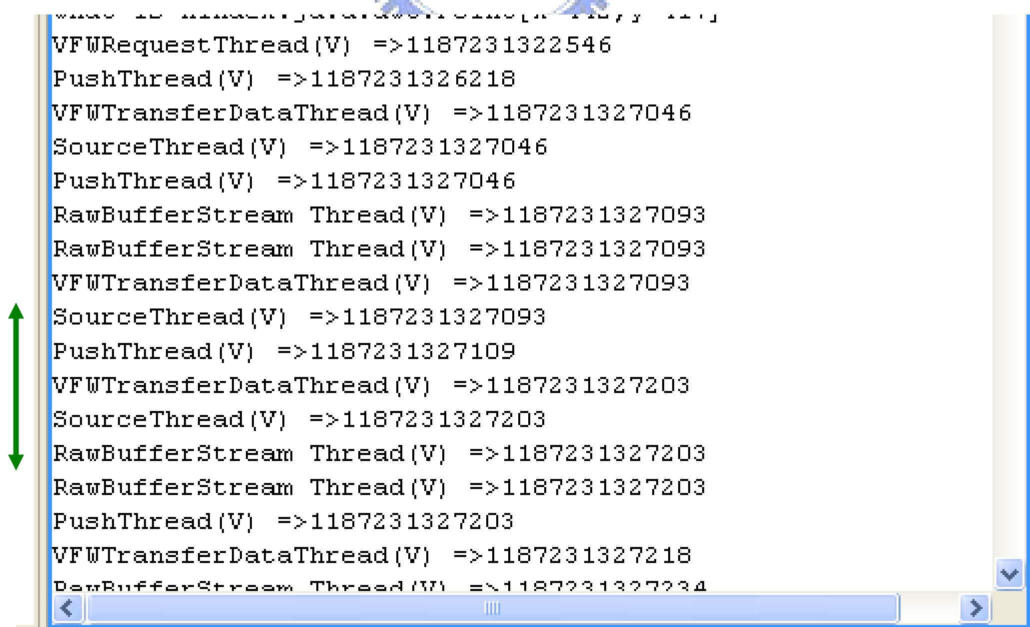
連續壓縮2張照片

```
VFWTransferDataThread(V) =>1187254989578
RawBufferStream Thread(V) =>1187254989609
SourceThread(V) =>1187254989609
VFWTransferDataThread(V) =>1187254989609
PushThread(V) =>1187254989625
RawBufferStream Thread(V) =>1187254989640
SourceThread(V) =>1187254989640
VFWTransferDataThread(V) =>1187254989640
RawBufferStream Thread(V) =>1187254989656
SourceThread(V) =>1187254989656
VFWTransferDataThread(V) =>1187254989656
RawBufferStream Thread(V) =>1187254989671
SourceThread(V) =>1187254989671
VFWTransferDataThread(V) =>1187254989671
PushThread(V) =>1187254989687
RawBufferStream Thread(V) =>1187254989703
SourceThread(V) =>1187254989703
RawBufferStream Thread(V) =>1187254989734
RawBufferStream Thread(V) =>1187254989734
```

(2)設置 10 秒時的情況



(3)設置 20 秒時的情況



(4)設置 40 秒時的情況

```
What is nIndex:java.awt.Point[x=442,y=417]
VFWRequestThread(V) =>1187232140796
PushThread(V) =>1187232145078
PushThread(V) =>1187232145093
VFWTransferDataThread(V) =>1187232145750
SourceThread(V) =>1187232145750
PushThread(V) =>1187232145750
RawBufferStream Thread(V) =>1187232145796
RawBufferStream Thread(V) =>1187232145796
VFWTransferDataThread(V) =>1187232145812
SourceThread(V) =>1187232145812
RawBufferStream Thread(V) =>1187232145828
RawBufferStream Thread(V) =>1187232145828
PushThread(V) =>1187232145937
VFWTransferDataThread(V) =>1187232145937
SourceThread(V) =>1187232145937
PushThread(V) =>1187232145953
VFWTransferDataThread(V) =>1187232145953
SourceThread(V) =>1187232145953
RawBufferStream Thread(V) =>1187232145953
```

自傳

韓孟潔，民國七十年生於屏東市。民國九十三年畢業於國立高雄應用科技大學電機工程學系，於民國九十四年八月進入國立交通大學通訊與網路科技產業研發碩士班就讀。民國九十六年八月取得碩士學位，論文題目是「多執行緒在多媒體串流的應用」。

研究興趣是 Java/C/C++程式開發，希望可以達到恣意揮灑的境界；生活興趣靜態方面:看電影、卡通、布袋戲，動態方面:籃球、棒球、游泳、田徑。