國 立 交 通 大 學

電信工程學系

碩 士 論 文

桌面訊息的變動偵測與壓縮傳送

Detection of Change of Desktop Content and Compress Transmission

研 究 生：蔡文賢

指導教授：張文鐘 教授

中 華 民 國 九 十 七 年 八 月

桌面訊息的變動偵測與壓縮傳送

Detection of Change of Desktop Content and Compress Transmission

研 究 生：蔡文賢　　Student：Wen-Hsien Tsai

指導教授：張文鐘　　Advisor：Wen-Thong Chang

國 立 交 通 大 學

電信工程學系

碩 士 論 文

A Thesis

Submitted to Department of Communication Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Communication Engineering

August 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年八月

桌面訊息的變動偵測與壓縮傳送

學生：蔡文賢　　指導教授：張文鐘 博士

國立交通大學電信工程學系碩士班

摘　　要

遠端桌面技術提供相當便利的功能。外地的使用者只需要使用此技術，便可操控遠處電腦，不受限於在本地端操作，因此只要有權限的使用者皆可使用。遠端桌面技術最重要的部分在於取得 Server 變動畫面，並傳送給 Client，因此，本論文針對這部分進行深入探討。在 Server 端，使用了 Windows API 函數來輔助我們取得可能變動範圍，和增加額外判斷區域，以這兩個機制縮小我們需比對的可能範圍，在比對後取得真正變動區域後會進行畫面資料的壓縮傳送。

論文另一部分是針對遠端桌面共有的問題：播放視訊檔案會造成網路流量大幅激增，因此，我們要加入 JPPEG 破壞性壓縮來處理畫面資料，而 JPEG 對影像部分較無影響，我們在傳送更新畫面時分為影像和非影像區兩類，當要更新的部分落入播放器區域會以 JPEG 破壞性壓縮；沒有落入播放器區域則以原定編碼方式傳壓縮資料。此論文是以 VNC 為平台，分為三部分：一、說明 Server 和 Client 間的運作流程；二、詳細探討 VNC Server 如何偵測桌面上真正變動畫面的區塊，和如何壓縮畫面資料傳送給 Client；三、針對畫面變動的視訊播放區域，加入 JPEG 壓縮，改善播放視訊時資料量爆增的問題，並提昇傳輸速度和品質。

Detection of Change of Desktop Content and Compress Transmission

Student：Wen-Hsien Tsai Advisors：Dr. Wen-Thong Chang

Department of Communication Engineering
National Chiao Tung University

ABSTRACT

Remote Desktop Technology provides a very convenient function. Outside users through this technology can control remote computer without restriction of operation at local side. Therefore, anyone with authorization can remote control their computer wherever. The most important part of remote desktop technology is that how to get change of desktop of Server and transfer them to Client. Consequencely, this thesis will focus on this issue. On Server side, we use Windows API to adminicle to get probably change region and add extra region. Through this two mechanism can reduce region that we need to compare. After comparing, we can get real change part and compress them for transmitting.

The other problem of remote desktop technology, it's playing video files will induce network flow huge increase. Therefore we add JPEG to compress image data.we divide into video and non-video part. When update part fall into video part, then we use JPEG. Otherwise we use original encoder to compress data.This thesis is based on Virtual Network Computing (VNC), it's composed of three parts: 1. Explain how VNC Server and Client work, 2. Detail discuss detects changed blocks on desktop, and how to compress these blocks to send to client. 3. We add JPEG algorithm which aimed at video playing areas to improve the problem that amount of data increased extremely when playing video files, and improve transfer speed and quality.

誌謝

在此最最最感謝的就是我的指導教授 張文鐘博士，感謝他提供我完善的資源，和一切所須的設備，讓我在二年碩士生涯中，學業課程和研究無後顧之憂，並在論文的盲點上，點出許多可能的解決方向。同時也謝謝 廖維國教授、黃仲陵教授、和范國清教授在口試時的指導，老師們真是我研究過程中的一盞明燈，讓我可以順利完成此論文。

接著要感謝實驗室裡的同學和學弟們，blue、明山、政達、峻權、振偉、夸克、志偉、盛如、honda、秉謙、建民、小瘋，感謝他們陪我渡過修課、趕作業報告、寫論文的這兩年，沒有他們在我失落或瓶頸時的支持和鼓勵，我可能沒辦法支撐下去，因此感謝你們給我這麼美好的回憶，期許我們一起成長前進。

感謝我的家人，父母親、姊姊，他們是我的支柱，在研究所兩年中，提供我一切生活所須，有如一股無形的力量支撐著我，讓我心無旁騖地研究，順利地完成論文。

最後感謝我的室友們，明山、維庭、冠勳，在兩年的共同扶持和幫助，也感謝所有認識的知心好友、運動的球友、和其他實驗室的朋友，謝謝你們陪我走過這段時光。謝謝！

誌於 2008.08 風城 交大

Wen-Hsien

目錄

表目錄

圖目錄

第一章 緒論

1.1 動機

在結合網路和電腦後，產生了相當便利的應用程式—遠端桌面技術！透過遠端桌面技術，我們能看到遠端電腦的桌面，即 Client 連線時，Server 會透過網路將螢幕上的畫面，完完整整地呈現給 Client。而 Client 可使用滑鼠和鍵盤直接操控，且 Client 端的硬體需求不高，因此，通過此技術可把一台即將淘汰的 486 轉換成能運行大型科學計算軟體的超級電腦。

實現此遠端桌面技術最關鍵部分在於取得 Server 的變動畫面，並將資料傳送給 Client 更新。Server 端可以定時透過 GDI 一連串程序，取得桌面完整畫面並和前一次畫面做比較，如此即可針對真正變動部分並傳送，但這種做法相當耗費 CPU 的效能和時間，從這個角度出發，本論文要先探討如何縮小比對的範圍。因桌面是由許多 process 組合而成，1. 我們透過 Windows 的 API，截取所有 Windows 送出的訊息，這些訊息會包含著通知 process 中元件需要更新畫面變化的區域；2. 接著再額外增加處理範圍，如將整個前景程式的視窗範圍皆加入判斷，目的在務必比對處理我們最關注的區域。以上機制是我們使用來取得可能的變動區域，達到縮小需要比對範圍的目的，只要我們比對前後畫面這些可能的區域範圍，即可取得真正有變動部分。

取得真正變動畫面範圍後，接著我們要探討另一問題：使用遠端桌面技術播放視訊檔案，畫面劇烈變動會造成網路傳輸量大幅激增，因此我要們加入破壞性壓縮 JPEG 來改善此嚴重問題。而畫面是由文字和圖片組合而成，如果皆對他們使用破壞性壓縮，因圖片高頻部分被捨去所造成的影響不大，但文字部分則會產生難以忍受的模糊，因此本論文另部分要對這些影像和非影像做區別壓縮。在取得真正變動畫面後，要先判斷 Server 端是否有開啟播放器，接著取得播放器在桌

面上的位置區域，只要真正變動的部分落在播放器裡就以 JPEG 壓縮，沒有落入的部分則以原定編碼方式，如此便可有效區分出影像和非影像部分，對他們進行差別壓縮。

1.2 研究背景

目前市面上有許多的遠端桌面程式，如 VNC、Microsoft 的 RDP、X Window…等，這些程式各有優缺點，本論文則是以 open source 的 VNC 為架構，VNC-Virtual Network Computing，為什麼選擇它呢？因為它有能力得到桌面的畫面，偵測出多視窗中何者變動、變動的範圍，將這些區塊的 pixel 資料加以壓縮，來降低資料量並傳送給 Client，而最讓人稱道的是：跨平台，只要 VNC Server 和 Client 兩邊皆遵守 RFB Protocol，就可透過 VNC Client 和 Windows、Linux、Unix、甚至 Mac 連線！

1.3 論文架構

本論文以 VNC 為主要架構，詳細說明一套遠端桌面系統的形成，因此分成幾個章節來詳細闡述。第二章先介紹市面上常見的遠端桌面程式，和他們之間的一些比較，接著說明要瞭解 VNC 所需要的 Win32 API；第三章是此論文的重心，詳細剖析 VNC Server 的整個運作結構、如何偵測實際變動畫面和如何傳送更新畫面的壓縮資料，第四章則講解 Client 的運作流程、變動畫面的更新、和如何透過滑鼠、鍵盤操控 Server；第五章將 Server 和 Client 間所有溝通訊息分為三大 Phase，分別詳細介紹和說明；第六章針對目前遠端桌面系統最大問題：播放視訊資料時，造成傳輸資料量爆增，提出改進方法：針對視訊畫面部分做 JPEG 壓縮，說明實作過程和各個數據的比較；第七章則是為本篇論文作出結論。

第二章 遠端桌面和相關背景知識

此章節一開始，先簡略介紹市面上常見的遠端桌面程式，接著說明 VNC 所遵循的 RFB Protocol，和要完全瞭解 VNC 所需的背景知識。2.1 節先簡單介紹 VNC、RDP、X Window 三個遠端桌面程式，並針對他們畫面更新的方式做綜合比較；2.2 節則是說明 VNC 所遵循的 RFB Protocol，它大略由 Display、Input、Pixel Format、溝通訊息四大部分所組成；2.3 節會將 VNC 內常用到的背景知識加以講解，如：Win32 程式建立的流程、視窗與視窗間溝通原理、畫面重畫訊息、和 Windows 的 API-Hook。

2.1 遠端桌面程式介紹

目前市面上有許多遠端桌面程式，在此就不一一介紹，僅選出較常用的三個來介紹，並淺要地分析他們的原理，和比較各項特點

2.1.1 VNC

Virtual Network Computing（VNC）是現行的遠端桌面連線系統之一，最初發展是在 1995 年由英國劍橋的 Olivetti & Oracle 實驗室發展出來，公開且免費的通訊協定，至今已發展多種版本，為跨平台的遠端桌面連線程式。

VNC 透過 Remote Frame Buffer (RFB) Protocol[1]控制遠端的電腦，VNC 為一套跨平台的應用程式，在操作端安裝 VNC Viewer 就是我們所說的 Client，在遠端的電腦安裝 VNC Server，允許多台 Clients 在同一個時間連接同一個 Server，另外，也可透過 JAVA Applet 使用網路瀏覽器連接到安裝 VNC Server 的電腦。VNC 的架構可以分為 VNC Server 和 VNC Client（Viewer），彼此溝通的通訊協定為 RFB Protocol。

●VNC Server：負責接收 Client 傳送過來的鍵盤或是滑鼠的輸入訊號後，並提供一套桌面分享機制，透過 TCP/IP 傳送畫面改變的資料到 Client。

●VNC Client：根據收到的資料，在將 Client 端顯示 Server 的畫面，並依照接收到的資料做更新。

●RFB Protocol：一個簡單的原則，將 Server 端某塊寬(W)、高(H)點陣圖(bitmap）的資料，放到螢幕指定的 X、Y 座標上。

2.1.2 RDP

Remote Desktop Protocol，簡稱 RDP [2]。RDP 是微軟根據 ITU T.120 協議系列所制定的一套未公開發表的數據通訊協議。透過網路連接 RDP Server 將應用程式顯示畫面傳送到 RDP Client，RDP Client 將滑鼠、鍵盤等輸入訊息傳送給 RDP Server。在畫面傳送上，是以命令(order)為操作方式，可以分為 primary order 和 secondary order，Primary order 主要的工作在於處理線條、矩形或是出現過的點陣圖和文字，Secondary order 則是傳遞首次出現的 bitmap 和文字。在終端服務的桌面圖形傳輸上，畫面中 bitmap 的傳送，可以說是主要的資料量。

RDP 的 bitmap 傳送方式有兩種方式：

1. RDP Server 先利用 secondary order 的 bitmap cache 將 bitmap 傳送到 RDP Client 暫存起來，隨後傳送一個 memory blt 將 cache 中 bitmap 資料，依據 cache id 和 cache index 取出指定的 bitmap，顯示在螢幕對應的位置上，主要畫面傳送都是利用這樣的方法。

2. RDP Server 直接傳送 bitmap 資料，並給予座標位置，RDP Client 收到後，直接將資料顯示在畫面上。傳遞的 bitmap 通常為視窗的外框或是工作列表邊緣線，因此，在畫面傳送上佔很小的比例，這部份的畫面更新方式稱為 bitmap updates。

2.1.3 X Window

X Window 系統（也常稱為 X11 或 X）是一種以點陣圖方式顯示的視窗系統[3]。最初是 1984 年麻省理工學院的研究，之後變成 UNIX、Linux 等作業系統所一致適用的標準化軟體工具套件及顯示架構的運作協定，經過二十多年的演進，現今已成為工業標準。X Window 與一般的作業系統不同，在設計時就是以 Client/Server 為理念，整個 X Window 可以分為幾個部份：X Server、X Client、X Protocol 和 X Library。

- X Server：負責控制顯示卡將影像畫在顯示器上，並且管理鍵盤和滑鼠的事件，產生視窗、對應視窗及刪除視窗，這部份要特別說明，與一般的 Client/ Server 名詞定義有點差異，X Server 定義為 Display Server，而應用程式端稱為 X Client。
- X Client：在 X Window 下的應用程式，要求特定的 X Server 作特定的動作。主要的工作為：1、向 X Server 提出需求，2、接收來自 X Server 的事件訊息，3、接收來自 X Server 的錯誤訊息。
- X Protocol：X Client 和 X Server 的通訊協定，定義 Requests、Reply、Error 和 Events。
- X Library：簡稱 X Lib，大部分 X window 上的應用程式以 X Library 來建立 GUI 元件，例如：按鈕（bottom）、目錄（menus）等等。

2.1.4 遠端桌面程式比較

根據先前介紹的遠端桌面系統，這節將對系統特性以及更新方式做個比較。

1. 畫面更新方式

Server 和 Client 之間的畫面更新方式可以分為以下幾種：

● RAW：所有的更新圖片，不經過壓縮編碼，每個 pixel 的資料依序傳送，可以想像資料量很大，假設傳送一張 640 x 480 大小的 16 位元高彩圖片，就需要 614.4K Bytes。VNC 有支援這樣的編碼方式機制。

●2D draw primitives：利用區塊填色的方法將圖形編碼，通常是一種非失真編碼的方式，VNC 畫面編碼主要是以這樣的方法。

●Low level graphic：除了圖形編碼的方法之外，還利用一些簡單的指令，例如：字型、多邊形、線條等等的指令呼叫 Client 繪圖，RDP 便是利用這樣的更新方式進行畫面更新。

●High level graphic：除了 2D draw primitives 和 Low level graphic，還支援視窗的建立和管理，X Window 所利用的 X Protocol 就是屬於這類。

2. Server/Client 訊息傳遞方式

Server 和 Client 的訊息傳遞方式主要可以分為兩種，Server Push 就是由 Server 主動決定何時傳送更新畫面到 Client；Client Pull 就是由 Client 向 Server 要求傳送更新，使用 Server Push 的系統，其畫面更新較為平順但是相對資料量較大，而 Client Pull 則是只有在使用者輸入訊息時，才會通知 Server 作畫面更新的傳送，但是其資料量較少。

瞭解了畫面編碼方式及系統更新方式，根據這些特點我們將針對這幾個系統做分析比較，說明如表 1.。[4]

表 2-1.遠端桌面系統比較

系統	畫面編碼方式	更新方式	壓縮方式	cache	license
VNC	Compressed Pixel Data	Client pull	2D draw primitives	Client frame buffer	GPL
RDP	Low level graphics	Server push	2D RLE	YES	proprietary
X window	High level graphics	Server push	none	No	GPL

2.2 RFB Protocol

VNC 的信令、溝通格式、編碼…等，完全是依照「RFB Protocol」所規定，因此，不論是任何版本的 VNC 皆須遵照此 protocol。RFB(Remote Frame Buffer)是一個定義遠端圖形用戶終端介面的簡單協定，因為它是用在 FrameBuffer 上，所以它適用於所有視窗作業系統和視窗程式，包括：X11、Windows 和 Mac。使用者端稱為 RFB Client，另一端則是 FrameBuffer 的源頭，可以是視窗系統和程式，稱為 RFB Server，如圖 1。

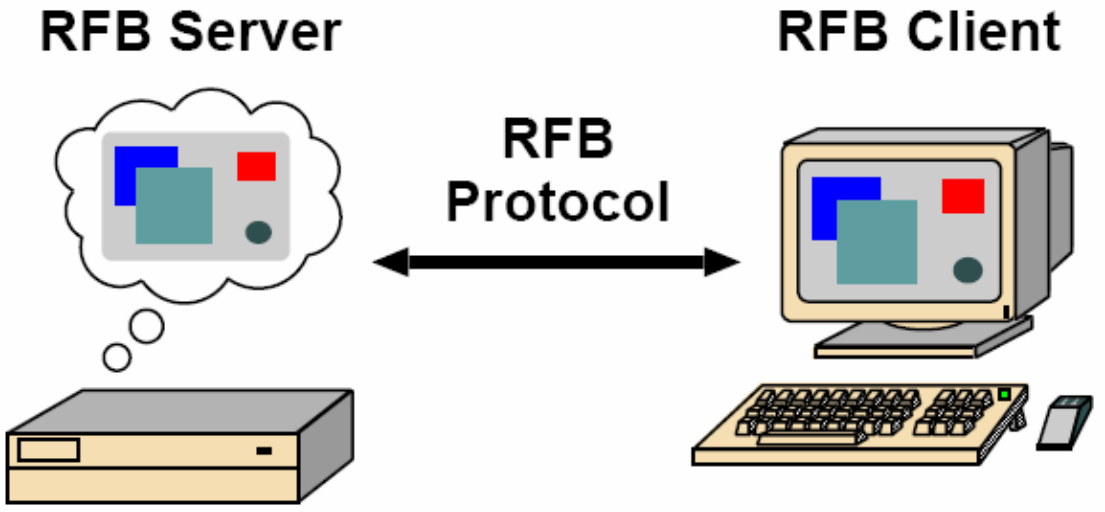


圖 2-1.VNC 系統架構

RFB Protocol 設計時考量的重點中，就強調 Client 端只需要相當少的需求，因此它可應用於廣泛的硬體上，目前市面上更出現相關的軟體，如 Smart VNC，讓手機也可以連上個人電腦，其方便性可見一斑。

2.2.1 RFB Protocol 組成要素

此小節要初步介紹構成 RFB 的幾大要素。

1. Display Protocol

Display Protocol 是以一個很原始且簡單的概念為基礎：「將 server Framebuffer 中，一個矩形所有的 pixel 資料，放在所屬 client 對應的位置上」。由許多的矩形組成一張完整的 Framebuffer，直覺上可能很沒效率，但深入探討後，當畫面只有一些小區塊變動而不是整張 Framebuffer，此時只需更新那區塊的矩形，加上其允許不同的編碼格式，提供了網路頻寬、使用端畫面更新和伺服端處理速度，相互間的取捨！

FrameBuffer 的變動，會促成 Server 送出一連串矩形的更新，但這個更新的機制是由 Client 為需求導向，即 Client 送出要求更新的訊息，Server 才會將資料送出。一般在本地端，更新 Framebuffer 是很快速的一個接著一個，Client 只需要持續的接收資料、解封包、更新本地端的畫面，接著再送出更新的需求，對於較次一級的 Client 或網路，當 Client 端更新速度跟不上 Server 端 Framebuffer 變化時，一些 Framebuffer 的暫態會被忽略掉。

2. Input Protocol

Protocol 的輸入端，是由標準的一組鍵盤，和多按鍵的點選裝置所組成！輸入的事件，無論使用者何時按了按鍵或移動裝置，都只由 Client 端傳送訊息給 Server 端。輸入裝置也可以是不標準的 I/O，如手寫板，他們的辨識引擎會自動產生鍵盤的對應事件，因此 Server 還是可以正確的判斷！

3. Pixel 資料表示

在 pixel 資料要被傳送時，RFB Client 和 Server 溝通中會包括格式和編碼，這個溝通是為了讓 Client 的工作簡單化而設計。底線是 Server 要提供 Client 想要的 pixel 資料格式，如果 Client 有能力處理許多不同的格式和編碼，它會選一個 Server 較容易產生的。

(1) Pixel 格式：個別顏色如何用 Pixel 值來表示，最常見的格式是 24-bit 或 16-bit，這些 bit 會直接被轉譯成紅、綠、藍的顏色。而 8-bit 的”colour map”，可以通過 mapping，把 pixel 值轉成 RGB 的顏色！

(2)編碼：一個矩形的 pixel 資料如何被傳送。每一個矩形的 pixel 資料，會先用一個標頭記錄這個矩形在螢幕中 X、Y 的位置、長、寬，和編碼的種類，而編碼資料會接在標頭之後被傳送出去。常用的編碼種類有 Raw、CopyRect、RRE、Hextile 和 ZRLE。一般都是使用 ZRLE、Hextile、CopyRect，因為他們提供最佳的壓縮效率！

4. 溝通訊息

RFB Protocol 可以應用在任何可靠的傳輸中，一般是在 TCP/IP 上。

這個 Protocol 總共會有三個階段：

(1)Handshaking Phase：用來確認 protocol 的版本和安全性的種類

(2) Initialization Phase： Client 和 Server 會互相傳送初始化的訊息

(3)Normal protocol interaction： Client 可以傳任何他想送的訊息，Server 會回傳結果！所有這些訊息都會以 message-type 為開端，接著才是訊息的資料！

Protocol 的訊息都是以以下幾種種類表示：U8、U16、U32、S8、S16、S32 各別是 unsigned 8、16、32bits 、signed 8、16、32bits，而所有多 bytes 都是以 big endian 來排序！PIXEL 則是一個 pixel 值所需的 bytes 數，即 BytesPerPixel。

2.3 Win32 API

因為 VNC Server 和 Client 中，有許多功能需借用 Windows 的函數來完成，如取得程式視窗的大小；畫面變動偵測，會用到訊息溝再來取得 Hook 偵測到的變動範圍；透過 GDI 將桌面的 Framebuffer 存成 RGB 的記憶體陣列…等，還有許許多多要借用 Win32 的 API，因此此節要介紹較重要的 API，其他部分則會在後續的章節陸續介紹。

2.3.1 視窗程式建立流程

此小節要介紹和一個很重要的概念：Handle，和完整 Win32 程式建立流程。

●Handle 概念

視窗程式全都是模組化的程式--每個視窗元件，例如選單、捲軸、按鈕等，都是一個獨立的個體，每個個體或元件都有自己的資料和函數，由於一個程式有可能會有多個相同種類的元件(例如一個程式裡有很多按鈕)，那麼在眾多相同的元件中，要決定控制哪個元件，就需要設定 Handle，來指明控制哪個元件，例如視窗程式的主視窗，就需要一個 Handler，來指明使用者現在正控制該程式的視窗，而非其他，故 Handler 的概念非常重要。

●Win32 程式建立流程[5]

```
int PASCAL WinMain(……)
{    HWND hwnd;    //宣告程式的 Handle
     MSG msg;          //宣告 MSG 結構來存放訊息
     WNDCLASS wc;//宣告類別結構

     InitWindow( …) ) //創建主窗口

     while (GetMessage(&msg, NULL, 0, 0))   //訊息迴圈
     {    TranslateMessage(&msg);          //解析 msg
          DispatchMessage(&msg);           //呼叫訊息處理函數
     }
}
```

每個程式都會有一個（只此一個）程式進入點（Entry Point），而 Windows API 程式的進入點為 WinMain。進入後會先宣告三個相當重要的參數，HWND、MSG、WNDCLASS，分別是程式最重要的 handler、溝通訊息的結構、和視窗程式的風格。這三個參數在往後程式運作過程中都佔有相當大的重要性。

接著在 InitWindow(…)中，會先對類別結構 WNDCLASS 做設定，然後調用 RegisterClass()對該類別進行註冊。因為每個視窗程式都有一些基本的屬性，如視窗邊框、標題欄文字、視窗大小和位置、滑鼠圖示、背景色、處理視窗的訊息函數名稱等等。註冊的過程也就是將這些屬性告訴系統，然後再調用 CreateWindow()函數創建出此視窗。這也就象你去裁縫店訂做一件衣服，先要告訴店老闆你的身材尺寸、布料顏色、以及你想要的款式，然後他才能為你做出一件讓你滿意的衣服。

以上初始化完成後，視窗建立大致上結束，接著會進入訊息迴圈，然後一直重覆截取訊息、解析、呼叫訊息處理函數，再回到截取…，直到收到使用者要結束時所發出的關閉訊息，才會離開此迴圈結束程式。

2.3.2 訊息迴圈

承接上節，此節詳細闡述訊息迴圈的過程。

Windows 應用程式可以接收以各種形式輸入的資訊，這包括鍵盤、滑鼠動作、記時器產生的訊息，也可以是其他應用程式發來的息等等。Windows 系統自動監控所有的輸入設備，並將其訊息放入該應用程式的訊息佇列中。

●GetMessage()

GetMessage()函數則是用來從應用程式的訊息佇列中按照先進先出的原則將這些訊息一個個的取出來，放進一個 MSG 結構中去。GetMessage()函數原型如下：[6]

```
BOOL GetMessage(
LPMSG lpMsg,            //指向一個 MSG 結構的指標，用來保存訊息
HWND hWnd,              //指定哪個視窗的訊息將被獲取
UINT wMsgFilterMin,     //指定獲取的主訊息值的最小值
UINT wMsgFilterMax      //指定獲取的主訊息值的最大值
);
```

GetMessage()將獲取的訊息複製到一個 MSG 結構中。如果佇列中沒有任何訊息，GetMessage()函數將一直空閒直到佇列中又有訊息時再返回。如果佇列中已有訊息，它將取出一個後返回。MSG 結構包含了一條 Windows 訊息的完整資訊，其定義如下：

```
typedef struct tagMSG {
HWND hwnd;          //接收訊息的視窗控制碼
UINT message;       //主訊息值
WPARAM wParam;      //副訊息值，其具體含義依賴於主訊息值
LPARAM lParam;      //副訊息值，其具體含義依賴於主訊息值
DWORD time;         //訊息被投遞的時間
POINT pt;           //滑鼠的位置
} MSG;
```

該結構中的主訊息表明了訊息的類型，例如是鍵盤訊息還是滑鼠訊息等，副訊息的含義則依賴於主訊息值，例如：如果主訊息是鍵盤訊息，那麼副訊息中則儲存了是鍵盤的哪個具體鍵的資訊。

GetMessage()函數還可以過濾訊息，它的第二個參數是用來指定從哪個視窗的訊息佇列中獲取訊息，其他視窗的訊息將被過濾掉。如果該參數為 NULL，則 GetMessage()從該應用程式的所有視窗的訊息佇列中獲取訊息。第三個和第四個參數是用來過濾 MSG 結構中主訊息值的，主訊息值在 wMsgFilterMin 和 wMsgFilterMax 之外的訊息將被過濾掉。如果這兩個參數為 0，則表示接收所有訊息。當 GetMessage()函數在獲取到 WM_QUIT 訊息後，將返回 0 值，程式將退出訊息迴圈。

●訊息處理函數

TranslateMessage()函數的作用是把虛擬鍵訊息轉換到字元訊息，以滿足鍵盤輸入的需要。DispatchMessage()函數的工作是把當前的訊息，發送到該應用程式對應的訊息處理函數去。

```
LRESULT CALLBACK WindowProc(          //訊息處理函數
HWND hwnd,                  //接收訊息視窗的控制碼
UINT uMsg,                  //主訊息值
WPARAM wParam,              //副訊息值
LPARAM lParam               //副訊息值      )
{    switch( uMsg ){
     case WM_KEYDOWN: …….              //擊鍵訊息
     case VK_ESCAPE: ……                //按下 ESC
     case WM_RBUTTONDOWN: ……           //滑鼠訊息
     case WM_PAINT:……                  //視窗重畫訊息
     case WM_DESTROY:……                //退出訊息
     Default：DefWindowProc(hWnd, uMsg, wParam, lParam); }
}
```

Windows 的訊息處理函數有一個確定的樣式,即這種函數的參數個數和類型以及其返回值的類型都有明確的規定。訊息處理函數的四個參數是由GetMessage()函數從訊息佇列中獲得 MSG 結構，然後分解後得到的。第二個參數 uMsg 和 MSG 結構中的 message 值是一致的，代表了主訊息值。程式中用 switch 語句來將不同類型的訊息分配到不同的處理程式中去。

值得注意的是，應用程式發送到視窗的訊息遠遠不止以上這幾條，像 WM_SIZE、WM_MINIMIZE、WM_CREATE、WM_MOVE 等這樣常常使用的訊息就有幾十條。為了減輕編寫程式的負擔，Windows 的 API 提供了 DefWindowProc()函數來處理這些最常用的訊息，調用了這個函數後，這些訊息將按照系統默認的方式得到處理。因此，在 switch_case 語句中，只須明確的處理那些有必要進行特別回應的訊息,把其餘的訊息交給 DefWindowProc()函數來處理，是一種明智的選擇,也是你必須做的一件事。

●結束訊息迴圈

當用戶按 Alt+F4 或單擊視窗右上角的退出按鈕，系統就向應用程式發送一條 WM_DESTROY 的訊息。在處理此訊息時，調用了 PostQuitMessage()函數，該函數會給視窗的訊息佇列中發送一條 WM_QUIT 的訊息。在訊息迴圈中，GetMessage()函數一旦檢索到這條訊息，就會返回 FALSE，從而結束訊息迴圈，隨後，程式也結束。

2.3.3 WM_PAINT

Windows 裡有各種功能的訊息，功能和格式各有不同，限於篇幅，這邊就不一一贅述，但遠端遠桌面系統最重要的便是得知畫面變動，進而壓縮這些區域傳送給 Client，因此此節要介紹和畫面更新最相關的訊息： “WM_PAINT “[7]。

●WM_PAINT 訊息

Windows 通過發送 WM_PAINT 訊息通知視窗訊息處理程式，視窗的部分顯示區域需要繪製。在 Windows 中，只能在視窗的顯示區域繪製文字和圖形，而且不能確保在顯示區域內顯示的內容會一直保留到程式下一次有意地改寫它時還保留在那裡。例如，使用者可能會在螢幕上移動另一個程式的視窗，這樣就可能覆蓋您的應用程式視窗的一部分。Windows 不會保存您的視窗中被其他程式覆蓋的區域，當程式移開後，Windows 會要求您的程式更新顯示區域的這個部分。

視窗訊息處理程式應在任何時刻都準備好處理其他 WM_PAINT 訊息，必要的話，甚至重新繪製視窗的整個顯示區域。在發生下面幾種事件之一時，視窗訊息處理程式會接收到一個 WM_PAINT 訊息：

1. 在使用者移動視窗或顯示視窗時，視窗中先前被隱藏的區域重新可見。
2. 使用者改變視窗的大小
3. 程式使用 ScrollWindow 或 ScrollDC 函式滾動顯示區域的一部分。
4. 程式使用 InvalidateRect 或 InvalidateRgn 函式刻意產生 WM_PAINT。

在某些情況下，顯示區域的一部分被臨時覆蓋，Windows 試圖保存一個顯示區域，並在以後恢復它，在以下情況下，Windows"可能"發送 WM_PAINT 訊息：

1. Windows 擦除覆蓋了部分視窗的對話方塊或訊息方塊。
2. 功能表下拉出來，然後被釋放。
3. 顯示工具提示訊息。

在某些情況下，Windows 總是保存它所覆蓋的顯示區域，然後恢復它。這些情況是：

1. 滑鼠游標穿越顯示區域。
2. 圖示拖過顯示區域。

程式應該組織成可以保留繪製顯示區域需要的所有資訊，並在 Windows 給視窗訊息函數發 WM_PAINT 訊息時才進行繪製。如果程式在其他時間需要更新其顯示區域，它可以強制 Windows 產生一個 WM_PAINT。

●Update Region

儘管視窗訊息處理程一旦接收 WM_PAINT 訊息之後，就準備更新整個顯示區域，但它經常只需要更新一個較小的區域。顯然，當對話方塊覆蓋了部分顯示區域時，情況即是如此。在擦除對話方塊之後，需要重畫的只是先前被對話方塊遮住的矩形區域。這個區域稱為“Invalid Region“或更新區域。正是顯示區域內無效區域的存在，才提示了 Windows 將一個 WM_PAINT 訊息放在應用程式的訊息佇列中。只有在顯示區域的某一部分失效時，視窗才會接受 WM_PAINT 訊息。

Windows 內部為每個視窗保存一個“繪圖資訊結構“，這個結構包含了包圍無效區域的最小矩形的座標以及其他資訊，這個矩形就叫做“無效矩形“，有時也稱為“無效區域“。如果在視窗訊息處理函數處理 WM_PAINT 訊息之前顯示區域中的另一個區域變為無效，則 Windows 計算出一個包圍兩個區域的新的無效區域，並將這種變化後的資訊放在繪製資訊結構中。Windows 不會將多個 WM_PAINT 訊息都放在訊息佇列中。

視窗訊息處理程式可以通過呼叫 InvalidateRect 使顯示區域內的矩形無效。如果訊息佇列中已經包含一個 WM_PAINT 訊息，Windows 將計算出新的無效矩形。否則，它將一個新的 WM_PAINT 訊息放入訊息佇列中。在接收到 WM_PAINT 訊息時，視窗訊息處理函數可以取得無效矩形的座標。通過呼叫 GetUpdateRect，可以在任何時候取得這些座標。

在處理 WM_PAINT 訊息處理期間，視窗訊息處理函數呼叫 BeginPaint 之後，整個顯示區域即變為有效。程式也可以通過呼叫 ValidateRect 函式使顯示區域內的任意矩形區域變為有效。如果這呼叫具有令整個無效區域變為有效，則目前佇列中的任何 WM_PAINT 訊息都將被刪除。

●WM_PAINT 訊息處理

```
case WM_PAINT：
hdc = BeginPaint(hwnd,&ps);
使用 GDI 函數
EndPaint(hwnd,&ps);
Return ;
```

在處理 WM_PAINT 訊息，必須成對地呼叫 BeginPaint 和 EndPaint。在呼叫完 BeginPaint 後，就開始呼叫 GDI 函數做一些畫面的處理。如果視窗訊息處理程式不處理 WM_PAINT 訊息，則它必須將 WM_PAINT 訊息傳遞給 Windows 中 DefWindowProc(內定視窗訊息處理函數)。

Windows 將一個 WM_PAINT 訊息放到訊息佇列中，是因為顯示區域的一部分無效。如果不呼叫 BeginPaint 和 EndPaint(或者 ValidateRect)，則 Widnows 不會使該區域變為有效。相反，Windows 將發送另一個 WM_PAINT 訊息，且一直發送下去。

2.3.4 Hook

因為遠端桌面程式最在意的便是畫面變動的區塊，因此如何取得這些區塊，就是這類程式相當重要的一部分。VNC 採用的方法是，監視任何和畫面相關的訊息，並將範圍記錄下來，再進一步處理，這些在後面章節會詳細探討，現階段我們要先研究 VNC 是如何取得想要的訊息---Hook[8]。

Hook 實際上是一個處理訊息的程式，屬於 Windows 的 API，通過使用者的啟動後，它可以針對某些特定動作所產生的訊息做監視和處理，比如：OS 發送 WM_PAINT 的訊息給某個應用程式、使用者敲擊鍵盤、移動滑鼠，這些就被分類為三種動作，因為 Windows 溝通都是透過訊息，因此每當特定的動作產生時，都會伴隨著訊息，在訊息還沒送達目的應用程式之前，Hook 就會先捕獲該訊息，亦即 Hook 前處理函數先得到控制權，此時對應 idHook 的前處理函數即可對該訊息加工，也可以不作處理而繼續傳遞該訊息，還可以強制結束訊息的傳遞。前處理函數會如何運作的部分我們在 3.4.1 節中會詳細再說明。

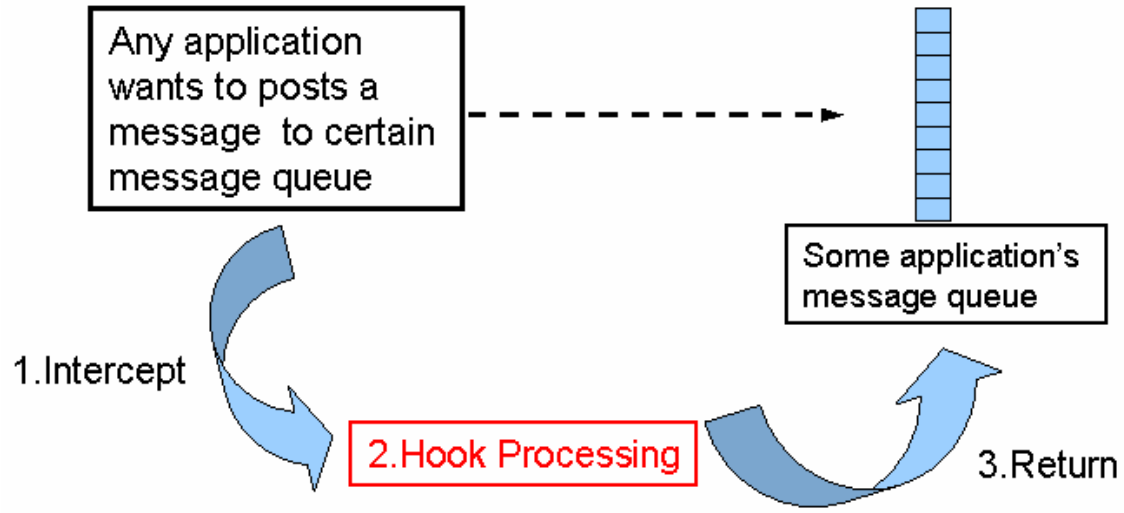


圖 2-2.Hook 示意圖

要實現 win32 的系統 Hook，必須調用 SDK 中的 API 函數 SetWindows-HookEx 來安裝這個 Hook 函數，這個函數的原型是：

```
HHOOK SetWindowsHookEx(
int idHook,          //要安裝的 Hook 類型 (參考下面的 IdHook)
HOOKPROC lpfn,       //Hook 前處理函數的指標，即攔截到指定系統訊息後
                     的前處理函數名稱,須定義在 DLL 中
HINSTANCE hMod,      //應用程式實例的控制碼
DWORD dwThreadId;    //要安裝 Hook 的 threadID ,指定被監視的 thread，如
                     果明確指定了某個 thread 的 ID 就只監視該 thread，此
                     時的 Hook 即為 thread hook；如果該參數被設置為 0，
                     則表示此 Hook 為監視系統所有 thread 的全局 Hook。
);
```

得到控制權的 Hook 前處理函數在完成對訊息的處理後，如果想要該訊息繼續傳遞，那麼它必須調用另外一個 SDK 中的 API 函數 CallNextHookEx 將訊息傳遞下去。Hook 預處理函數也可以通過直接返回 true 來丟棄該訊息，並阻止該訊息的傳遞。每種類型的 Hook 是由系統來維護，最後安裝的 Hook 放在鏈的開始，而最早安裝的 Hook 放在最後，也就是後加入的先獲得控制權。Hook 在使用完之後需要用 UnHookWindowsHookEx()卸載，否則會造成麻煩。

Windows 總共提供了對應 13 種動作的 Hook，這裡僅列出一般較常用的 9 種，旁邊是要觸發 Hook 對應動作的說明。

●常用 idHook 參數整理如下：

```
WH_CALLWNDPROC      //視窗 hook，當系統向目標視窗發送訊息時將觸發
WH_CALLWNDPROCRET //視窗 hook，當視窗處理完訊息後將觸發
WH_CBT              //當 Windows 啟動、產生、釋放（關閉）、最小
                      化、最大化或改變視窗時都將觸發此事件
WH_GETMESSAGE       //當往訊息佇列中增加一個訊息時將觸發此 hook
WH_JOURNALRECORD    //記錄 Hook，可以用於記錄滑鼠和鍵盤操作，木馬
                      程式可以使用此 Hook 竊取受控方在螢幕中敲入
                      的密碼
WH_KEYBOARD         //當敲擊鍵盤時將觸發此 hook
WH_MOUSE            //當有滑鼠操作時將觸發此 hook
WH_MSGFILTER        //訊息過濾 hook
WH_SYSMSGFILTER     //系統訊息過濾 hook
```

VNC Server 在程式中啟用了三個 Hook，分別是上面 WH_CALLWNDPROC、WH_GETMESSAGE、WH_SYSMSGFILTER，並且在 SetWindowsHookEx() 函數中，第四個參數設為 0，代表是監視全局的執行緒，即所有的程式執行時，只要符合以上三個 idhook 監視動作，Hook 皆會搶先一步處理這些訊息。因此 VNC Server 可以由這些訊息中，得到訊息的種類，如果是和畫面變動有關的訊息，比如：WM_PAINT，便可由訊息中的參數 hwnd，得知是何視窗可能發生變動，並記錄他們的大小範圍，接著進一步做比對和處理。

第三章 VNC Server

此章要細部探究 VNC Server 的工作原理，說明一套遠端桌面程式是如何成形的。遠端桌面程式一定需要接受 Client 的要求連線，進一步透過某種機制來互相溝通，且這類程式最主要目的就是取得 Server 的畫面並操控它，因此也需要一套完整偵測變動畫面方法和處理資料並傳送的流程。因此，在 3.1 節會就 VNC Server 整個完整架構先進行初步介紹，對整個邏輯有點概念；3.2 節中說明 Server 是如何不斷接收不同 Client 的連線要求；3.3 節討論從 Server 和 Client 一開始連線成功的初始化、兩者如何互相溝通，和有哪些溝通訊息的封包格式，都會在這說明；3 .4 節就是整篇論文最精華也最重要的部分，說明 VNC Server 透過兩種方法取得可能畫面變動區域：一、先透過 Hook 取得如整個程式的最小化還原時、點選選單、播放影片時的拖曳區和時間跳動…等可能變動畫面的訊息，由這些訊息所屬的視窗基本結構取得初步變化的區域大小，二、再由 VNC 自行額外加入全螢幕、前景程式…等的視窗範圍，因視窗程式工作區的資訊變動，可能是由應用程式自行產生，Hook 無從取得這些訊息，因此 VNC 自行將這些區域範圍也加入。由以上兩種方式取得最後可能變動範圍後，在最新資料的 mainbuffer 和前一狀態的 backbuffer 中，詳細比對這些可能範圍，取得真正有變動的區域資料並壓縮，再進一步傳送給 Client。

3.1 VNC Server 運作流程和架構

此章在先以流程圖大概描述 VNC Server 整個程式的運作流程，接著再以示意圖的方式，詳細說明每個區塊的功能，和他們之間的溝通方式。至於每個區塊的細節部分會在後續章節詳細探討。

●系統運作流程

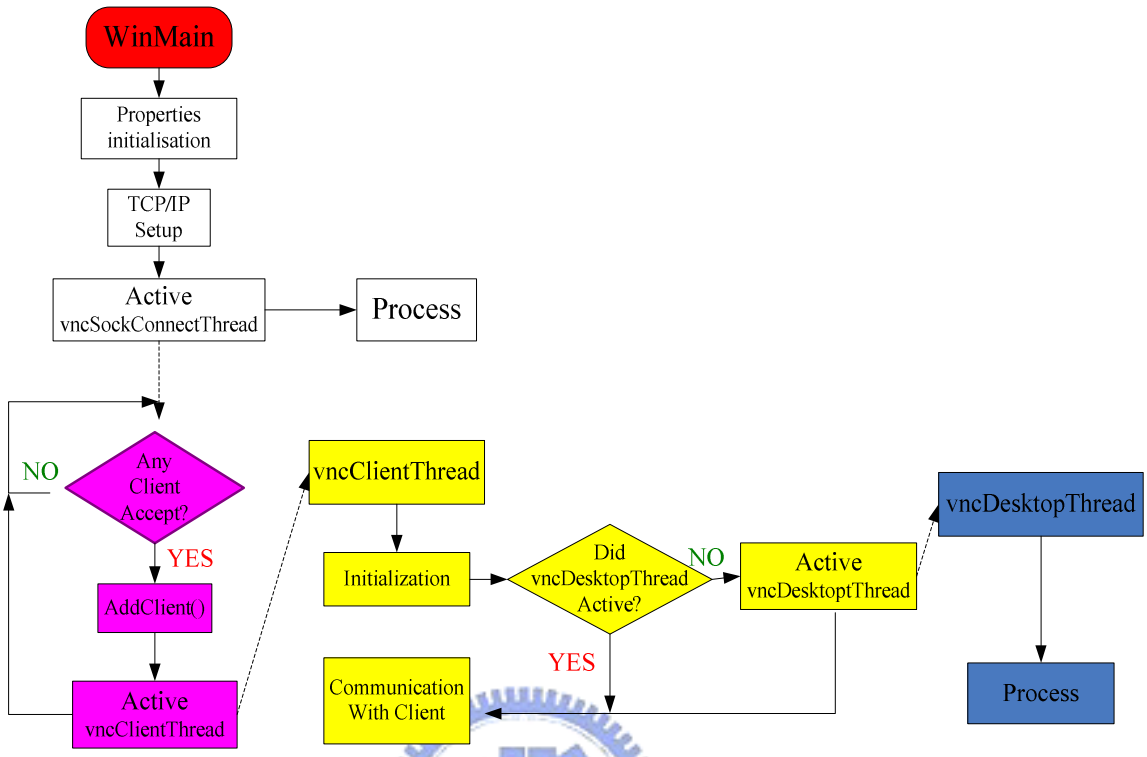


圖 3-1. VNC Server 流程圖

上圖 3 是 VNC Server 完整的流程圖，它主要由四個區塊所組成，分別是主程式和三個獨立的 Thread，他們會在運作中依序被啟動。Server 一開始由 WinMain 進行一些選項的初始，接著建立 TCP/IP 和 Client 溝通訊息封包，如 Create Socket、Bind、Listen，以上完成後就會啟動 Server 的第一隻 vncSockConnect Thread(上圖紫色區塊)，啟動後主程式即交由 vncSockConnectThread 做後續的處理，主程序隨即回去等待處理，使用者會對工具列上 Server 程式做的動作。vncSockConnectThread 的功能用來監聽是否有 Client 要求連線。當有 Client 傳送要求連線時，vncSockConnectThread 會將此 Client 加入，並開啟第二隻 vncClientThread(上圖黃色區塊)專責處理所有 Client 的溝通訊息，接著再回去繼續監聽。當 vncClientThread 開啟後會先和 Client 進行初始化，完成後偵測是否已有第三隻 vncDesktopThread(上圖深藍色區塊)的存在，若沒有則啟動它，之後回去處理 Client 送來的訊息。而 vncDesktopThread 則是專責處理畫面變動，當有畫面變動，且 Client 有要求更新，則將變動的畫面

資料傳送給 Client。這些 Thread 的細節在之後的每一節中有詳盡的說明。

●系統架構示意圖

下圖 4 是 VNC Server 整體的架構和運作流程，主要是由三隻 thread 所建構而成的，他們各司其職，在接收到 Client 發出的訊息，做出相對應的處理，直到 Client 離開連線，或關掉 Server 程式！這裡先初步介紹它們的功能，細節會在後續的章節中完整地說明。

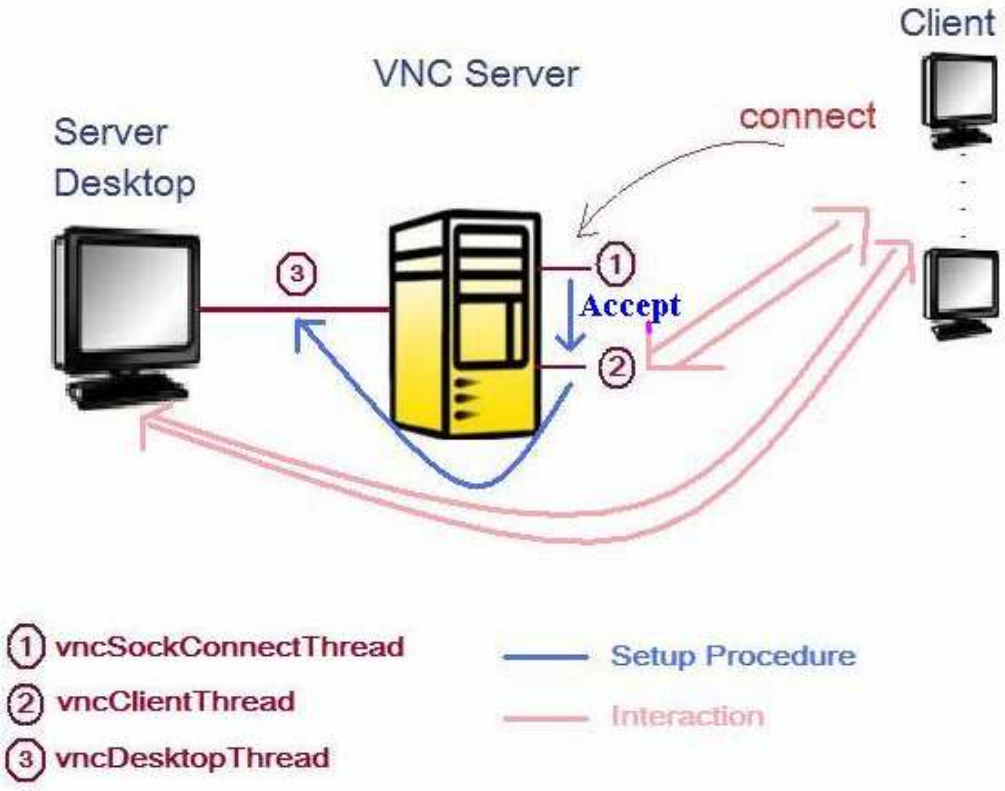


圖 3-2.VNC Server 系統架構

1. **vncSockConnectThread**：用來接洽所有 Client 的連線，當成功連線後，會開啟 vncClientThread 來接續後面的事項，接著回去繼續監聽其他 Client 連線要求。

2. **vncClientThread**：處理一切 Client 傳送過來的訊息，訊息可能是要設定編碼、要求畫面更新、游標移動、或敲擊鍵盤…等，它會針對不同的要

求做出不同的動作。

3. vncDesktopThread：偵測 Server 端桌面 Framebuffer 的實際變動區域，壓縮這些畫面，並依照固定的標頭和資料格式傳送給 Client。這部分是整個 Server 的核心所在，也是最複雜的地方，後面花最多篇幅詳細的講解。

●運作流程

Server 程式一開啟，要先檢查是否已有 Server 啟動，因為 VNC 不允許兩個 Server 同時存在。接著初始化一些必要的基本設定，如登入密碼、Port、View Only…等。設定完後會啟動 vncSockConnectThread 去建立完整的 TCP/IP 連線流程，Create socket、Bind、Listen，再來就是等待 Client 送出連線要求才會有進一步的動作。當 Client 要求連線時，vncSockConnectThread 會和 Client 進行三方交握(Three-Way-Handshaking)，Accept 後，Server 接著啟動 vncClientThread，而 vncSockConnectThread 則回到接收前狀態，繼續等待其他 Client 的連線。

vncClientThread 方面，一開始會先和 Client 進行版本的溝通和密碼的確認，當這些都正確無誤，它才會去啟動第三隻最重要的 vncDesktopThread。啟動之後，vncClientThread 就開始負責所有從 Client 送過來的訊息，因為 VNC 互相溝通的訊息格式都規範在 RFB Protocol 中，只要傳送過來的格式正確，它都能夠做出對應的處理，以維持兩端正常的運作。

vncDesktopThread 則是負責所有畫面的部分，包括存取本地端 Framebuffer 的初始化、設定傳輸更新訊息的封包格式、啟動 Hook、偵測可能畫面變動、辨別實際變動區塊…等，並將這些資料依照 Client 要求的編碼種類，和 RFB Protocol 中規定的標頭格式編碼後，傳送給 Client，因此這隻 thread 的重要性不

言可喻。 以上就是完整的運作流程。

3.2 vncSockConnectThread

此 Thread 是負責接收所有 Client 連線的要求，一開始說明基本的連線設定，後面會再說明不同 Client 要求連線時，Server 要如何應對。

3.2.1 TCP/IP 連線建立

Server 端程式一執行，便會先進行基本的設定，如密碼、是否 disable input、和設定功能表上選項的勾選。以上處理完，為了讓 VNC Client 可以對 Server 提出連線要求，Server 端必須接著進行 TCP/IP 的設定：Create Socket、Bind 和 Listen，之後會啟動 Server 的第一隻 Thread：vncSockConnectThread，透過設定好的 port 等待 Client 的連線要求。

```
// Create the listening socket
if (!m_socket.Create())
        return FALSE;

// Bind it
if (!m_socket.Bind(m_port, server->LoopbackOnly()))
        return FALSE;

// Set it to listen
if (!m_socket.Listen())
        return FALSE;

// Create the new thread
m_thread = new vncSockConnectThread;
if (m_thread == NULL)
        return FALSE;

// And start it running
return(((vncSockConnectThread *)m_thread)->Init(&m_socket, server));
```

圖 3-3.TCP/IP 連線建立

3.2.2 vncSockConnectThread 啟動

為了讓 Server 可以和一般網路程式一樣，接收一個 Client 後還可以繼續監聽是否有新的 Client 要求連線，VNC Server 將 TCP/IP 最後一個函數：Accept()，放在此 Thread 中，當 vncSockConnectThread 啟動後，隨即會進入一無窮迴圈等待 Client。一旦成功連線，Accept()函數會回傳一 VSocket 的類別，其中會記錄著 accept_socketid，後續和此 Client 資料的封包傳送和接收，皆是透過此

accept_socketid。Accept()完成後會呼叫 AddClient(new_socket, FALSE, FALSE)函數，去啟動另一獨立的 vncClientThread 專責和此 Client 溝通訊息，而 vncSockConnectThread 會再回到迴圈，繼續等待下一個連線要求，直到 Server 結束。以上即是 VNC Server 能夠持續接收許多 Client 連線的機制。

```
// Code to be executed by the thread
void *vncSockConnectThread::run_undetached(void * arg)
{
    log.Print(LL_STATE, VNCLOG("started socket connection thread\n"));

    // Go into a loop, listening for connections on the given socket
    while (!m_shutdown)
    {
        // Accept an incoming connection
        VSocket *new_socket = m_socket->Accept();
        if (new_socket == NULL)
            break;

        // Successful accept - start the client unauthenticated
        m_server->AddClient(new_socket, FALSE, FALSE);
    }
    log.Print(LL_STATE, VNCLOG("quitting socket connection thread\n"));

    return NULL;
}
```

圖 3-4.vncSockConnectThread

值得注意的是，vncSockConnectThread 在接收了不同 Client 連線要求後，會啟動對應但不完全相同的 vncClientThread，因為在 Server 和 Client 傳送和接收封包，是以 accept_socketid 當橋樑，因此每隻 vncClientThread 都要記錄著各別的 accept_socketid，Server 才有能力和所有 Client 溝通，分辨出此封包是由哪個 Client 送進來，應該傳送畫面資料封包到哪個 Client。

3.3 vncClientThread

此 Thread 是專責和 Client 做溝通，一切由 Client 送過來的訊息，如：畫面更新、敲擊鍵盤、移動或敲擊滑鼠、設定 Pixel_Format…等訊息，都是由此 Thread 負責解析，進而做相對應的處理。在第五節中會詳細說明訊息接收後的處理程序。

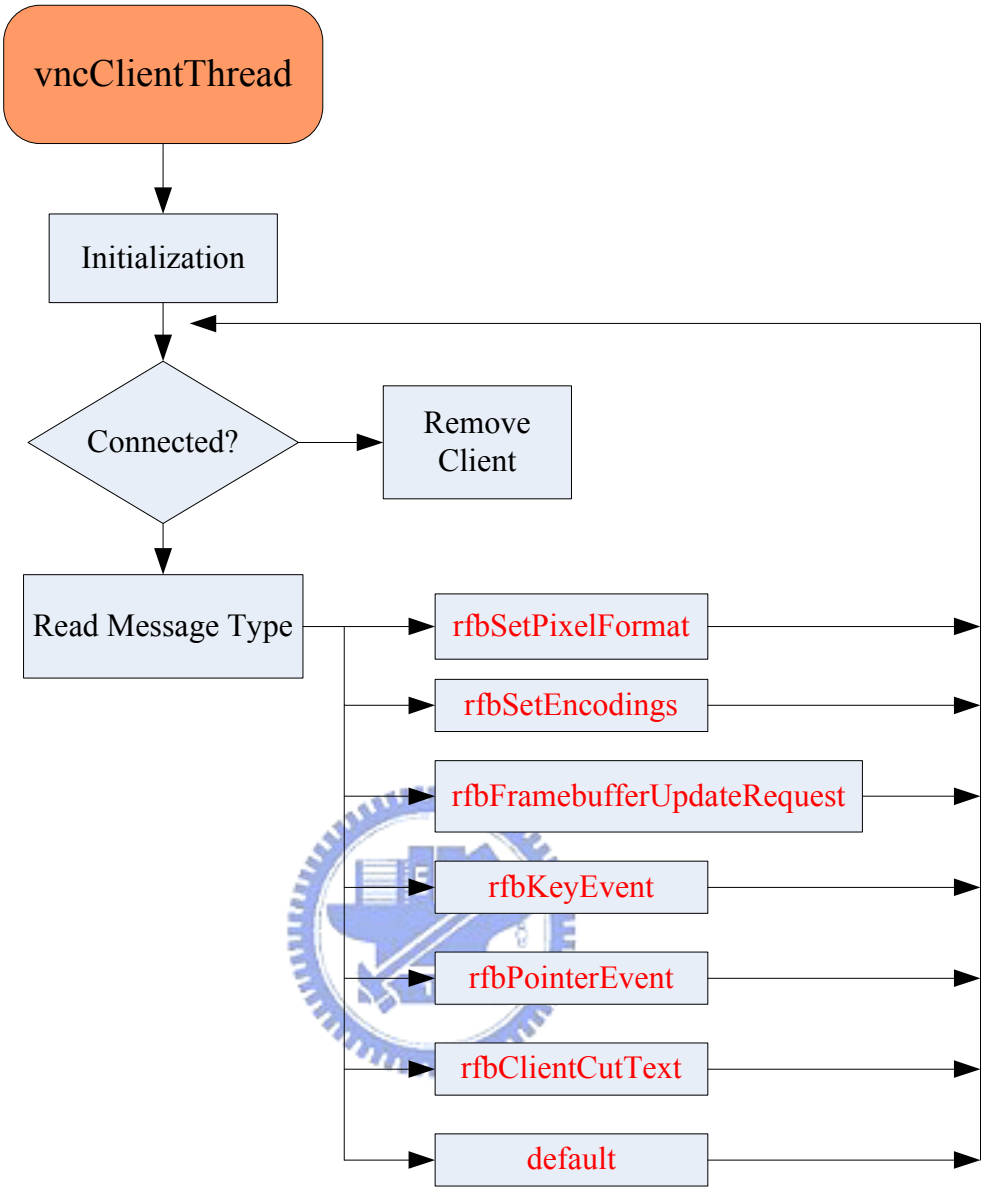


圖 3-5.vncClientThread 流程圖

3.3.1 vncClientThread 的 Initialization

Thread 一啟動，馬上開始 RFB Protocol 中的 Handshaking Phase，進行版本和安全性種類的確定，再進一步檢查密碼和認証是否通過，一切完成後接著判斷 vncDesktopThread 是否已啟動，若沒有則啟動它，啟動後繼續設定 mainbuff 和 backbuff，這兩塊 buffer 在後續傳送資料時會用到，最後才會開始 Server 和 Client 的初始化訊息溝通。

```
// LOCK INITIAL SETUP
// All clients have the m_protocol_ready flag set to FALSE initially, to prevent
// updates and suchlike interfering with the initial protocol negotiations.

// GET PROTOCOL VERSION
if (!InitVersion())
{
    m_server->RemoveClient(m_client->GetClientId());
    return;
}
log.Print(LL_INTINFO, VNCLOG("negotiated version\n"));

// AUTHENTICATE LINK
if (!InitAuthenticate())
{
    m_server->RemoveClient(m_client->GetClientId());
    return;
}
```

圖 3-6.Handshaking Phase

●InitVersion()

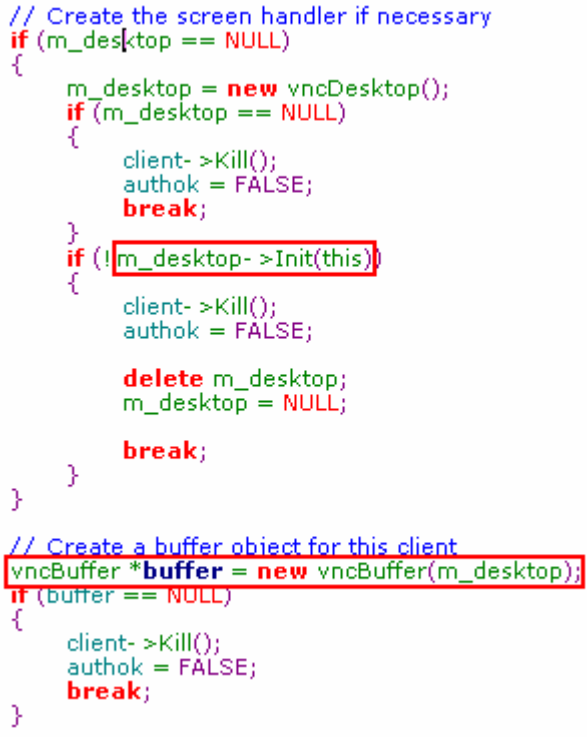
一開始會由 Server 傳送 Protocol Version 給 Client，讓 Client 知道 Server 最高可以支援的 RFB Protocol 版本，而 Client 收到後會回傳訊息，告知實際是用哪一個 protocol！但它不能夠要求高於 Server 的 protocol version。這個機制是為了使雙方都可以向下支援。目前發布的 RFB 版本有 3.3，3.7，3.8。新增進的編碼格式不會改變 protocol 的版本，因為 Server 可以忽略它不懂得的編碼。這部分在第五章會再詳細說明。

●InitAuthenticate()

在 Protocol Version 被決定後，雙方必須溝通連線安全性的種類，和密碼認証。這部分在第五章會再詳細說明。

●啟動 vncDesktopThread

版本、安全性種類、密碼認證完成後，代表 Server 已和 Client 建立完整的溝通管道，兩邊可以按照 RFB Protocol 規定進行下一步的動作。vncClientThread 一開始會先接收 ClientInit 訊息，判斷是否准許其他 Client 同時和 Server 連線。接著，vncClientThread 會開啟畫面更新和傳送資料最核心的 Thread—vncDesktopThread。



```
// Create the screen handler if necessary
if (m_desktop == NULL)
{
	m_desktop = new vncDesktop();
	if (m_desktop == NULL)
	{
		client->Kill();
		authok = FALSE;
		break;
	}
	if (!m_desktop->Init(this))
	{
		client->Kill();
		authok = FALSE;

		delete m_desktop;
		m_desktop = NULL;

		break;
	}
}

// Create a buffer object for this client
vncBuffer *buffer = new vncBuffer(m_desktop);
if (buffer == NULL)
{
	client->Kill();
	authok = FALSE;
	break;
}
```

圖 3-7.vncDesktopThread and SetBufffer

開啟 vncDesktopThread 之前，vncClientThread 會先判斷 Server 是否已有啟動 vncDesktopThread，因為同時間可以有很多個 vncClientThread 和 Server 溝通，但只需要一隻 Thread 來偵測畫面變動，因此若已有啟動則返回；沒有的話就去啟動它，至於啟動後 vncDesktopThread 的後續程序，我們會在下一節詳細探討。

●Frmaebuffer 設定

vncDesktopThread 啟動後，會去設定許許多子項，讓 VNC Server 透過一些變數，就可以得到目前 Framebuffer 的狀況。接著 vncClientThread 宣告 vncBuffer *buffer = new vncBuffer(m_desktop)，此類別中的 CheckBuffer()函數會依照得到的 Framebuffer 大小和屬性，建立 mainbuff 和 backbuff，並將目前桌面畫面資料存到兩個 buff 中。畫面產生變動時，只要把最新的變動範圍存入 mainbuff 中，再將這些範圍和舊資料的 backbuff 比對，即可得到真正變動區塊，這部分後面會詳細說明。

以上處理完，vncClientThread 會將 vncDesktopThread 得到的 Frmabuffer 的狀況，存放在 ServerInit 訊息中傳送給 Client，讓 Client 能夠在本地端開一個和 Server 相同大小屬性的 Viewer 視窗。接著 vncClientThread 會進入一個無窮迴圈，專責處理由 Client 送過來的訊息，並做相對應的處理。

3.3.2 Client to Sever 訊息的種類和處理

一切初始化流程結束後，vncClientThread 便會進入訊息處理迴圈，等待 Client 傳送訊息過來，並解析訊息和處理。在第六章會詳細介紹這些訊息，和 Server 如何處理他們。如果 Client 要傳送未定義的訊息，需確定 Server 是否有支援。下表 2 是 Protocol 中已訂定的幾個訊息。

表 3-1.Client to Server messages

Number	Name
0	**SetPixelFormat**
2	**SetEncodings**
3	**FramebufferUpdateRequest**
4	**KeyEvent**
5	**PointerEvent**
6	**ClientCutText**

以上即是 Client 可能傳送給 Server 的所有訊息種類，當一個訊息處理完，vncClientThread 會再回到迴圈啟始處，等待 Client 送過來的訊息，因此，vncClientThread 有如 Server 端的接待員，負責接洽 Client 的所有要求。

3.4 vncDesktopThread

接下來這節是整篇論文核心，說明 VNC Server 要如何獲得確切畫面變動區塊並壓縮傳送到 Client 端。在 3.4.1 節中介紹 vncDesktopThread 為了後續的工作，一開始先做了哪些初始化。3.4.2 節則是說明 Thread 如何取得初步變動範圍，

他們是由 Hook 處理和畫面相關訊息後送出的。3.4.3 節針對 Hook 沒辦法取得的畫面區塊進行補強，再額外加入比對的範圍。3.4.4 節一開始比對 3.4.3 節中獲得的最後範圍，將真正有變動的部分壓縮處理後傳送給 Client。

3.4.1 vncDesktopThread 啟動

vncDesktopThread 是由 vncClientThread 所啟動，啟動後馬上進行一連串的初始化，這些設定是為了設置許多參數，在後續運作中會用到，因此此節來說明這些函數細節。

```
// Initialise the Desktop object
KillScreenSaver();
if (!InitDesktop())
    return FALSE;
if (!InitBitmap())
    return FALSE;
if (!ThunkBitmapInfo())
    return FALSE;
if (!SetPixFormat())
    return FALSE;
EnableOptimisedBlits();
if (!SetPixShifts())
    return FALSE;
if (!SetPalette())
    return FALSE;
if (!InitWindow())
    return FALSE;
// Add the system hook
  !SetHook(
    m_hwnd,
    RFB_SCREEN_UPDATE,
    RFB_COPYRECT_UPDATE,
    RFB_MOUSE_UPDATE
    )
```

圖 3-8.Initialise the Desktop object

●KillScreenSaver()

此函數會停用螢幕保護程式。

●InitBitmap()

此函數主要功能是為了取得 Server 端桌面的狀況，供後續變數使用，如目前使用的解析度和維度大小…等，細部分為五大步驟：

```
BOOL vncDesktop::InitBitmap()
{
    // Get the device context for the whole screen and find it's size
1.  m_hrootdc = ::GetDC(NULL)   //If this value is NULL, GetDC retrieves the DC for the entire screen
    if (m_hrootdc == NULL)
        return FALSE;

2.  m_bmrect.left = m_bmrect.top = 0;
    m_bmrect.right = GetDeviceCaps(m_hrootdc, HORZRES);
    m_bmrect.bottom = GetDeviceCaps(m_hrootdc, VERTRES);

    // Create a compatible memory DC
3.  m_hmemdc = CreateCompatibleDC(m_hrootdc);
    if (m_hmemdc == NULL) {
        return FALSE;
    }

    // Create the bitmap to be compatible with the ROOT DC!!!
    m_membitmap = CreateCompatibleBitmap(m_hrootdc, m_bmrect.right, m_bmrect.bottom);
    if (m_membitmap == NULL) {
        return FALSE;
    }

    // Get the bitmap's format and colour details
    int result;
4.  m_bminfo.bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    m_bminfo.bmi.bmiHeader.biBitCount = 0;
    result = ::GetDIBits(m_hmemdc, m_membitmap, 0, 1, NULL, &m_bminfo.bmi, DIB_RGB_COLORS);

    // Henceforth we want to use a top-down scanning representation
    m_bminfo.bmi.bmiHeader.biHeight = - abs(m_bminfo.bmi.bmiHeader.biHeight);

    // Is the bitmap palette-based or truecolour?
5.  m_bminfo.truecolour = (GetDeviceCaps(m_hmemdc, RASTERCAPS) & RC_PALETTE) == 0;

    return TRUE;
} ? end InitBitmap ?
```

圖 3-9.InitBitmap()

1. 取得整個桌面的 Device Context[1]。

2. 由 Device Context 得到目前桌面的長和寬。

3. 在記憶體中建立一塊和桌面相同的 DC，因為後續會用到，接著將桌面 DC 的資料，由 CreateCompatibleBitmap()函數轉換成 Bitmap 的類別，存放在記憶體中，後續會再用到。

4. 用 GetDIBits()函數，可將此 m_membitmap 這個 Bitmap 類別中的 BITMAPINFO，全部填到 m_bminfo.bmi 中。

5. 判斷 Server 端是否有使用調色盤，若沒有，則 truecolour 為非零值。

[1] 當 Windows 程式在螢幕、印表機或其他輸出設備上畫圖時，它並不是將圖直接輸出到設備上，而是將圖由 Device Context 表示的”邏輯意義”，映射到"顯示平面"上去。Device Context 是 Windows 內部的一種資料結構，它包含 GDI 需要的所有關於顯示平面狀況的描述欄位，包括相連的設備和各式各樣的狀態資訊。在螢幕上畫圖之前，Windows 程式從 GDI 獲取 Device Context Handle，每一次調用一個 GDI 輸出函數時，它就會把這個 Handle 傳回給 GDI。如果沒有有效的 Device Context Handle，則 GDI 不會做任何的繪圖動作。Device Context 使 GDI 擺脫設備限制的過程中發揮了重要的作用。

●ThunkBitmapInfo()

此函數會判斷 Server 端的色彩品質，若不是 16bits 或 32bits，會調整一些 m_bminfo.bmi 中的參數，但目前大部分的電腦都是 32bits，因此通常會直接跳到下個函數。

●SetPixFormat()

將剛得到的桌面資訊，如：trueColour、bigEndian、Width、Height、bitsPerPixel、depth、biBitCount，複製到 m_scrinfo 這個結構中，之後需要某些參數，就統一從 m_scrinfo 讀取。

●EnableOptimisedBlits()

```
void
vncDesktop::EnableOptimisedBlits()
{
    // Create a new DIB section ***
    HBITMAP tempbitmap = CreateDIBSection(m_hmemdc, &m_bminfo.bmi,
                                          DIB_RGB_COLORS, &m_DIBbits, NULL, 0);
    if (tempbitmap == NULL) {
        m_DIBbits = NULL;
        return;
    }

    // Replace old membitmap with DIB section
    m_membitmap = tempbitmap;
}
```

圖 3-10.EnableOptimisedBlits()

此函數包含了相當深入的 Bitmap 問題，但最重要的功能是在建立一塊和桌面相同大小的記憶體區塊，它會以 m_bminfo.bmi 中的長、寬、bitsperpixel 來設定，當呼叫 CreateDIBSection()函數後，m_DIBbits 即是此記憶體區塊的指標，回傳值 tempbitmap 即為 DIBSection，接著會撤換掉先前的 m_membitmap。

建立 DIB Section 為的是能夠使用快速的 Blt 函數複製桌面 DC 上的資料，後續只要簡單的程序(請參閱 3.4.4 節的 Capture Screen)，就能將桌面上畫面像素資料迅速複製到 m_DIBbits 中，且在 vncClientThread 呼叫 CheckBuffer()函數時，就將 mainbuff 指向 m_DIBbits，以上就是記錄桌面 Framebuffer 的 mainbuff 記憶體位置。

●SetPixShifts()

用來設定 ServerInit 中的 Shift 值。

16bits：redMask = 0x7c00; greenMask = 0x03e0; blueMask = 0x001f

32bits：redMask = 0xff0000; greenMask = 0xff00; blueMask = 0x00ff

●SetPalette()

如果是 Server 端是全彩，則跳過此函數；反之，建立調色盤，供 Client 使用。

●InitWindow()

此函數會建立一個 win32 程式，但不顯現在螢幕上，VNC Server 會用它來作為和 Hook 溝通的橋樑，因為每個 win32 程式都有 handle 來辨識，也有一個訊息佇列來和 Windows 或其他視窗程式溝通。因此，只要 Hook 截取到有用的訊息，它會處理此訊息，並再以訊息的方式存放在此隱形程式的訊息佇列，VNC Server 只要定期來檢查它的訊息佇列中是否有訊息，即可判斷畫面變動的狀況。後面會以例子做說明。

●SetHook()

```
// Add the CallWnd hook
hCallWndHook = SetWindowsHookEx(
                    WH_CALLWNDPROC,          // Hook in before msg reaches app
                    (HOOKPROC) CallWndProc,  // Hook procedure
                    hInstance,               // This DLL instance
                    0L );                    // Hook in to all apps
// Add the GetMessage hook
hGetMsgHook = SetWindowsHookEx(
                    WH_GETMESSAGE,           // Hook in before msg reaches app
                    (HOOKPROC) GetMessageProc,     // Hook procedure
                    hInstance,               // This DLL instance
                    0L );                    // Hook in to all apps
    // Add the GetMessage hook
hDialogMsgHook = SetWindowsHookEx(
                    WH_SYSMSGFILTER,         // Hook in dialogs, menus and scrollbars
                    (HOOKPROC) DialogMessageProc, // Hook procedure
                    hInstance,               // This DLL instance
                    0L   );                  // Hook in to all apps
```

圖 3-11.SetHook

接續在第二章後半部介紹的 Hook，這邊要說明它是如何運作，和做了什麼工作。Hook 是在此函數中啟用的，VNC Server 總共會啟用三種 Hook：WH_CALLWNDPROC、WH_GETMESSAGE、WH_SYSMSGFILTER，他們對

應的前處理函數分別是 CallWndProc、GetMessageProc、DialogMessageProc，這三個函數只會在符合情況的訊息出現時被呼叫。這三個前處理函數被呼叫後，因為格式不同，都會將傳入的參數加以處理，最後，他們都會再用這些參數呼叫 Hookhandle()函數，Hookhandle()函數會依照不同種類的訊息做不同的處理。

```
// Hook procedure for CallWindow hook
LRESULT CALLBACK CallWndProc(int nCode, WPARAM wParam, LPARAM lParam)
{
   CWPSTRUCT *cwpStruct = (CWPSTRUCT *) lParam;

   HookHandle(cwpStruct->message, cwpStruct->hwnd, cwpStruct->wParam, cwpStruct->lParam);
   // Call the next handler in the chain
   return CallNextHookEx (hCallWndHook, nCode, wParam, lParam);
}

// Hook procedure for GetMessageProc hook
LRESULT CALLBACK GetMessageProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    MSG *msg = (MSG *) lParam;
    // Only handle application messages if they're being removed:
    if (wParam & PM_REMOVE)
    {
        // Handle the message
        HookHandle(msg->message, msg->hwnd, msg->wParam, msg->lParam);
    }
    // Call the next handler in the chain
   return CallNextHookEx (hGetMsgHook, nCode, wParam, lParam);
}
// Hook procedure for DialogMessageProc hook
LRESULT CALLBACK DialogMessageProc(int nCode, WPARAM wParam, LPARAM lParam)
{
    MSG *msg = (MSG *) lParam;
    // Handle the message
    HookHandle(msg->message, msg->hwnd, msg->wParam, msg->lParam);
    // Call the next handler in the chain
   return CallNextHookEx (hGetMsgHook, nCode, wParam, lParam);
}
```

圖 3-12.Hook Procedure

假設以 WM_PAINT 訊息為例，當有一個視窗出現無效區域(如縮小、還原)需要重畫時，Windows 會發送一 WM_PAINT 訊息給此視窗，此動作一產生後，Server 所啟用的 Hook-WH_CALLWNDPROC 就會先動作，Hook 會先呼叫它對應的前處理函數 GetMessageProc()，之後將訊息種類、所屬 hwnd、訊息的副消息當參數呼叫 HookHandle()函數。HookHandle()函數一開始由訊息種類可以判斷是 WM_PAINT，再用 Windows 的 SDK 函數 GetWindowsRect()得知此訊息所屬視窗的位置大小，將此位置大小的值用訊息的方式 Post 到， 在 Server 端 Initialization 中啟用的隱形程式的訊息佇列，因此 Server 只要從訊息佇列中取出訊息就可知道某個區域可能即將發生畫面變化，接著對這個視窗的範圍做進一步

比對處理即可。

3.4.2 vncDesktopThread 畫面處理流程

vncDesktopThread 一開始的初始化完成後，就會進入處理畫面的無窮迴圈裡，總共有三大部分，一、取得 Hook 偵測可能的範圍，二、進一步增加額外判斷範圍，三、將最後可能變動範圍進行比對並傳送真正變動區塊。下圖為此 Thread 的流程圖。

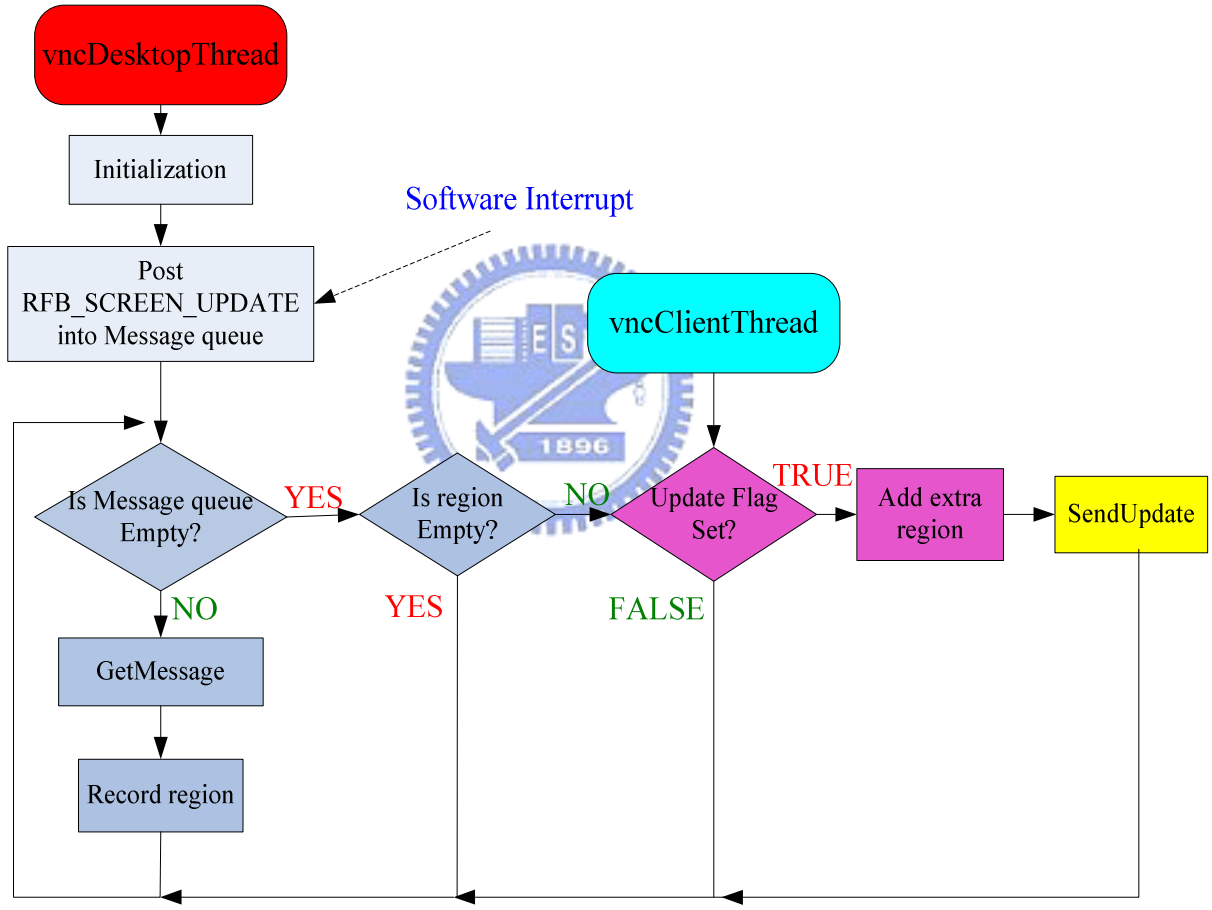


圖 3-13.vncDesktopThread 流程圖

這個迴圈相當繁瑣， 大致上是由三個部分所組成。

一、取得 Hook 偵測到可能的範圍

1. vncDesktopThread 初始化後，剛進入迴圈時，可能尚未有如 4.3.1 節中啟用的三種 Hook 動作發生，因此，他們對應的前處理函數不會送來任何可

能變動的訊息到隱形程式的訊息佇列，因此佇列中沒有任何訊息。接著繼續判斷 region 中是否有記錄任何可能的變動區塊大小，若沒有，代表目前沒有任何需要更新的區域，隨即再回到迴圈中，等待訊息並加以處理。

2. 一切開始正常運作後，當一產生 4.3.1 節中啟用三種 Hook 對應的動作後，對應 Hook 的前處理函數會處理可能和畫面變動相關的訊息，處理完後會再以訊息方式(RFB_SCREEN_UPDATE)， Post 到 Initialization 時啟動的隱形程式的訊息佇列。vncDesktopThread 進入迴圈後會先判斷訊息佇列中是否有新訊息，當它發現有訊息時，會取出此訊息判斷，如果訊息是 RFB_SCREEN_ UPDATE 就表示是由對應動作 Hook 的前處理函數處理過後傳過來的，接著 vncDesktopThread 由此 RFB_SCREEN_ UPDATE 消息的副消息參數，可以得到可能變動的大小，並將它記錄到 rgncache(Window 的 Region 類別)中，如下圖。接著再回到佇列中看是否有新的訊息，若有，則再取出並和剛剛記錄的做聯集，直到訊息佇列中都沒訊息為止。

```
// Now wait on further messages, or quit if told to
if (!GetMessage(&msg, m_desktop->Window(), 0,0))
    break;

// Now switch, dependent upon the message type recieved
if (msg.message == RFB_SCREEN_UPDATE)
{
    // An area of the screen has changed
    RECT rect;
    rect.left =     (SHORT) LOWORD(msg.wParam);
    rect.top = (SHORT) HIWORD(msg.wParam);
    rect.right = (SHORT) LOWORD(msg.lParam);
    rect.bottom = (SHORT) HIWORD(msg.lParam);

    rgncache.AddRect(rect);
}
```

圖 3-14.GetMessage and Record

3. 訊息佇列清空後，代表現階段沒有任何新的畫面變動，Thread 會趁此空檔趕緊去做後續處理，因此再判斷 region 不為空後，會再判斷旗標 Update Flag 是否被設為 TRUE，它代表 Client 有要求畫面更新，若有要求才會進一步處理。若 Client 沒有要求，則 Flag 為 False，因此 Thread 會將剛記錄的 region 保留著，直接返回剛剛 vncDesktopThread 訊息處理迴圈，等待下次記錄完並清空訊息佇列後再一起處理。這是因為 VNC 的整個架

構是 Client Pull，Update Flag 是由 vncClientThread 所控制，在它收到 Client 的畫面更新要求後，才會將 Update Flag 設為 TRUE，因此主動權在 Client 端，只有在 Client 有要求後，Server 才能做後續的處理。

二、增加額外判斷範圍

在 region 不為空且 Update Flag 被設為 TRUE 時，表示剛剛 vncDesktop-Thread 已發現可能有畫面變動，並記錄著這些區塊的座標大小在 region 中，而且當 Client 也要求要更新畫面時，Thread 就要接著後續的處理。為了避免一些應用程式，可能不是透過 Windows 以畫面重畫的訊息機制更新畫面，而是自行處理內部工作區域的變動，因此在 4.3.1 節中啟用的三種 Hook 無從取得可能變動範圍。VNC Server 在這部分，是以自行增加額外範圍來應對，它總共提供三種方式：Poll FullScreen、Poll Foreground Window、Poll Window Under Cursor。

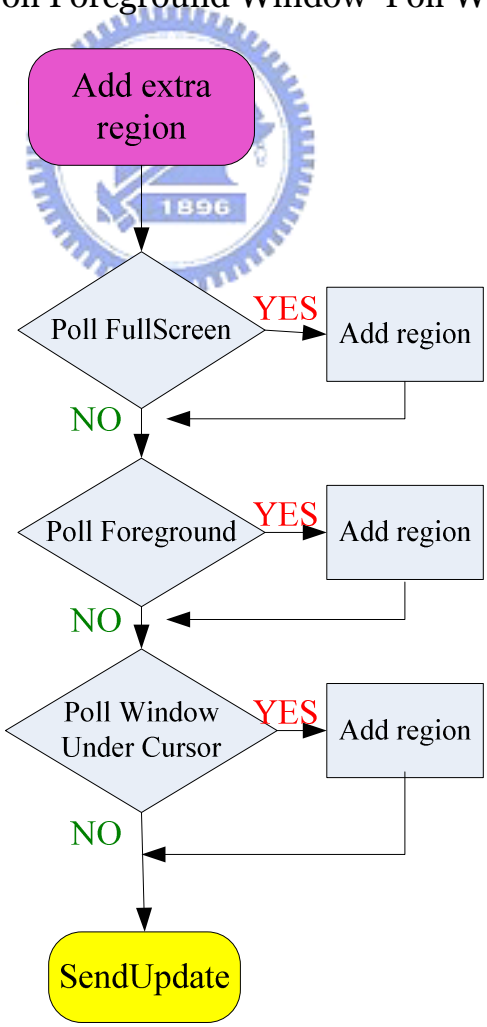


圖 3-15.Add extra region 流程圖

●Poll Full Screen

當使用者勾選了這個選項，代表為了確保整個螢幕都要更新到，不希望失去某些畫面，此時 VNC Server 會將螢幕範圍拆成四等分，每觸發一次都會額外再加入一個等分，如下圖。因此，採取這樣的方式會稍稍增加資料的處理量。VNC Server 也可以每次都加入全螢幕的大小範圍，但為了克服網路頻寬和 Computing Power 的問題，它採取的是經過四次的觸發流程，才會更新完整個畫面。

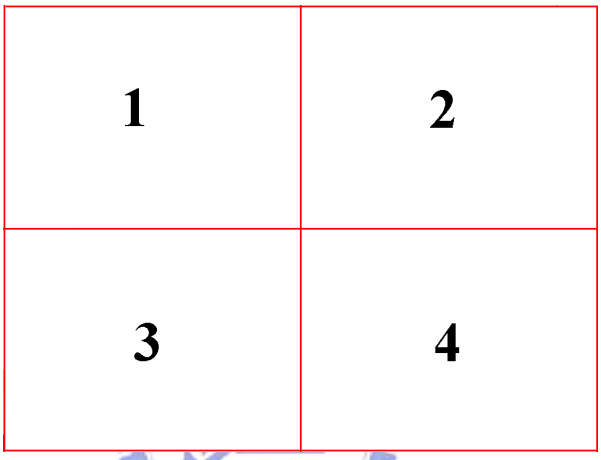


圖 3-16.Poll Full Screen

因為每一次清空訊息佇列中的訊息得到初步範圍後，再進到 Add extra region 時，只會加上螢幕的四分之一，加入的方式如下圖程式碼，用 m_pollingcycle 記住狀態，然後以 m_pollingcycle 計算這次要加入的是哪一塊範圍，並記錄在矩形結構 rect 中，再將這個座標大小與 vncDesktopThread 前端取得的初始範圍做聯集，接著送去 SendUpdate()函數處理後再回到迴圈，再等下次清空佇列裡的訊息，才有機會再進 Add extra region 加上另一塊四分之一的畫面，因此要重覆四次才能更新完成。

```
RECT rect;
rect.left = (m_pollingcycle % 2) * m_qtrscreen.right;
rect.right = rect.left + m_qtrscreen.right;
rect.top = (m_pollingcycle / 2) * m_qtrscreen.bottom;
rect.bottom = rect.top + m_qtrscreen.bottom;
m_changed_rgn.AddRect(rect);
m_pollingcycle = (m_pollingcycle + 1) % 4;
```

圖 3-17.Pollingcycle 狀態

假設在更新一塊後，後續可能因為都沒有任何變動，因此 vncDesktopThread 會一直卡在等待佇列中新的訊息產生，那剩下三塊的資料，有可能會延遲相當長的時間才能完成，因此，Poll FullScreen 是種定量卻不定時的機制，對於這點，我們可以設一個類似計時器的功能，在一定時間內觸發一次，讓他能夠持續的更新完整的畫面，這是值得我們改進的地方。

●Poll Foreground Window

如果使用者勾選了 Poll Foreground Window 選項，表示他要目前使用中的前景應用程式視窗範圍，也加進去剛取得的初步範圍，避免應用程式內部工作區的漏掉。它會先以 hwnd = GetForegroundWindow()取得前景程式的 HWND，再以 GetWindowRect(hwnd, &rect)將前景程式的視窗座標大小存入矩形的結構 rect，接著將此大小加入到 vncDesktopThread 得到的初步範圍。

●Poll Window Under Cursor

至於 Poll Window Under Cursor 這個選項，和上面的類似，這個則是要加入目前游標下方，應用程式視窗大小的範圍。它會先以 GetCursorPos(&mousepos)取得滑鼠的位置存在 mousepos 中，再以 hwnd= WindowFromPoint(mousepos)取得滑鼠位置下方的視窗程式座標和大小，接著也是加入到 vncDesktopThread 得到的初步範圍中。

三、將最後可能變動範圍進行比對並壓縮傳送真正變動區塊

當這三個選項都判斷完後，偵測變動範圍的部分已告一段落，由 Hook 和 VNC 額外加入範圍，以上兩種方式取得最後可能變動範圍後，會呼叫 SendUpdate 函數，在最新資料的 mainbuff 和前一狀態的 backbuff 兩個 buffer 中，詳細比對這些可能範圍，取得真正有變動的區域資料並壓縮，再進一步傳送給 Client，

這部分比較直觀，在 3.4.4 小節會詳細介紹。

3.4.3 變動畫面偵測

在上一小節中，說明了 VNC Server 偵測變動範圍的整個詳細過程，在這一節以實際例子說明讓大家能有更深刻了解。這小節用兩個例子來說明，一、訊息佇列的管理機制：說明當使用者操控鍵盤、滑鼠時，Windows 會得知這些動作消息並做出對應回應，而我們在 4.3.1 節中啟動的三種 Hook，就是截取伴隨著這些動作所產生的訊息，當此訊息是和畫面相關，則對應 Hook 的前處理函數會將變動區塊的座標大小，Post 隱形程式的訊息佇列裡，接著 vncDesktopThread 再從佇列中取出加以記錄。二、Extra Update Region 管理：由訊息佇列管理機制中取得初步範圍後，為了確保所有變動區域都有取得，VNC 額外加入三種機制，Pull FullScreen、Poll Foreground Window、Poll Window Under Cursor，以這三種機制再額外將重要的範圍加進來判斷處理。

●訊息佇列的管理機制：

下圖為一開始的狀態，接著使用者操控滑鼠點選小算盤的圖示，將他還原顯示在桌面上，如圖。

圖 3-18.初始狀態

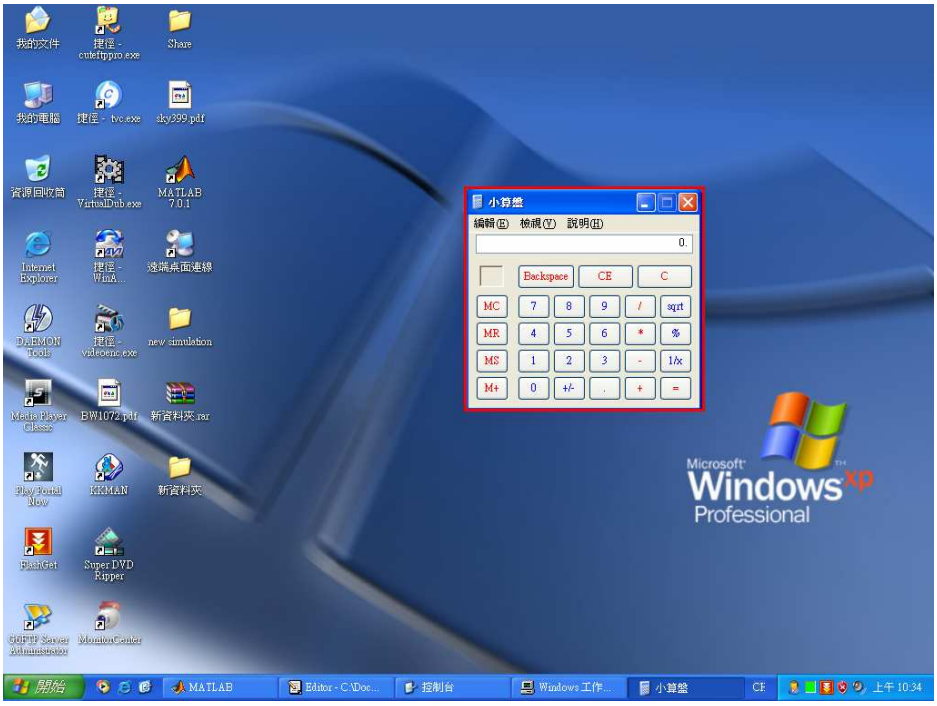


圖 3-19.還原小算盤

當使用者欲操控滑鼠將小算盤還原時，OS 內部能夠計算出小算盤應該在哪個地方被還原出來，並以 WM_PAIT 的訊息送給小算盤的 WndProc 訊息處理函數，此時 Server 所啟用的 Hook-WH_CALLWNDPROC 會先動作，它會偵測到 OS 發送 WM_PAINT 訊息給小算盤程式，請它重畫在螢幕上，此時對應的前處理函數 CallWndProc()會先處理 WM_PAINT 訊息，處理後會將小算盤的視窗大小以訊息方式(RFB_SCREEN_ UPDATE)，Post 到隱形程式的訊息佇列。當 vncDesktopThread 進入迴圈後會先判斷訊息佇列中是否有新訊息，當它發現有訊息時，會取出此訊息，接著由此消息的副消息參數得到應用程式的大小，並將它記錄到 rgncache (Window 的 Region 類別)中。因此，rgncache 裡面會記載著此應用程式的座標和大小，接著 vncDesktopThread 再回到迴圈開頭判斷是否還有未處理的消息，若沒有，就會繼續後面的流程。在這個 case 中，前半部分已先由 Windows 的訊息和 Hook 的搭配，取得完整應用程式的視窗範圍，在後面的 Add extra region 中只是再增加範圍進去，以上即是訊息佇列的管理機制。後續會再接著呼叫 SendUpdate 去做其他的處理。

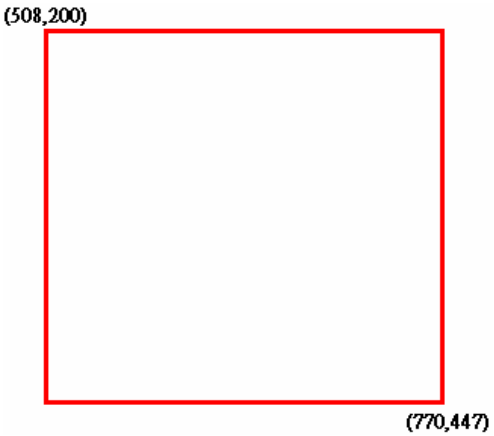


圖 3-20.rgncache 記錄的座標大小

●Extra Update Region 管理：

接著用另一個較複雜的例子來說明，當 VNC Server 遇到其他狀況時，它要如何應對。假設 Client 連入時 Server 已在播放視訊檔案，如下圖。

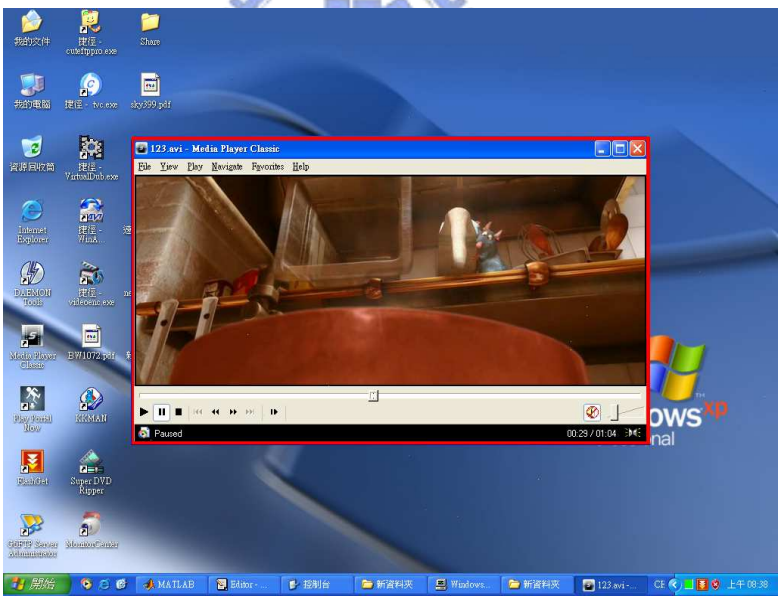


圖 3-21.Server 初始狀況

一般而言，此時，4.3.1 節中啟用的三種 Hook，應該有能力偵測到整個 mplayer 的視窗大小，再進行後續的處理，但此時的訊息佇列中卻只會有兩種訊息和此播放器有關的，如下圖紅框。

圖 3-22.mplayer 的變動訊息

因為拖曳區和時間都有獨立的 handle，所以 Server 初始化時所啟用的 Hook-WH_CALLWNDPROC 能偵測到發給他們的 WM_PAINT 訊息，因此處理後，隱形程式的訊息佇列中會有上圖畫紅框的範圍大小，但為何 vncDesktopThread 無法獲得播放區域的範圍呢？常理來講，應該要可以偵測到畫面變動(工作區域)、拖曳區、時間這三個地方，但是，播放器的原理是它取得視訊檔案後，在內部解碼後直接呼叫 DirectDraw，將圖片資料送到顯示卡的記憶體秀出，或由 Windows GDI 在指定區域將解碼後的圖片畫上去，並不是以訊息的方式通知 mplayer 請它重畫，因為若是以訊息的方式通知重畫，可能造成播放的延遲，影響觀看的品質。這部分就是應用程式內部工作區，不一定會以訊息方式處理，因此，我們必須要在 Add extra region 中再加入必要範圍。

假設一開始訊息佇列中有兩個訊息，一個是拖曳區，另一個是時間，迴圈起始處判斷佇列中有訊息，會先記錄第一個訊息中的範圍到 rgncache。

圖 3-23.rgncache 狀態 1

回到迴圈後發現訊息佇列中還有訊息，Thread 會再記錄到 rgncache，並移

除掉第二個訊息，等所有訊息都記錄完畢，就會進入迴圈下一個部分。因此，Hook 得到初步可能的變動範圍只有如圖所示。

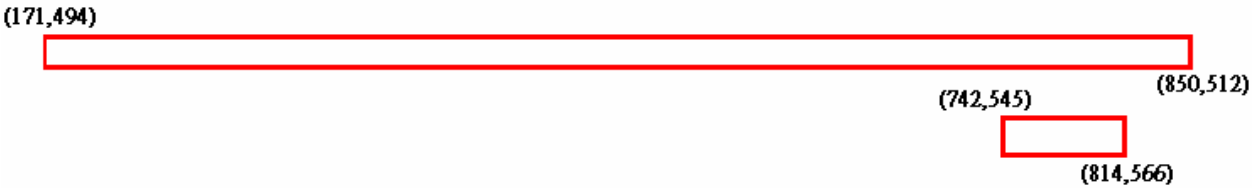


圖 3-24.Hook 偵測到的初步變動範圍

在 vncDesktopThread 處理訊息佇列中的訊息後，取得了初步的範圍，可能因為許多不同狀況中，如上圖，mplayer 在播放影片時，因為畫面更新的來源並不是透過訊息，因此 Hook 對最重要的畫面區域沒辦法偵測到，因此，VNC Server 在 Update Flag 設為 True 後，接著後續的 Add extra region 這部分額外加入三個選項：Poll FullScreen、Poll Foreground Window、Poll window under cursor，針對不同狀況加入額外的範圍，確保雙方工作正常，接著為這三種狀況詳細說明。

●若勾選了 Poll Full Screen

若勾選 Poll FullScreen 了，則是要確保所有部分都要更新到，Thread 每次會將螢幕四分之一等分大小範圍加進去，當第一次觸發時，會加上 1 的區塊，第二次則換加 2 的區塊…以此類推，四個為一個循環。

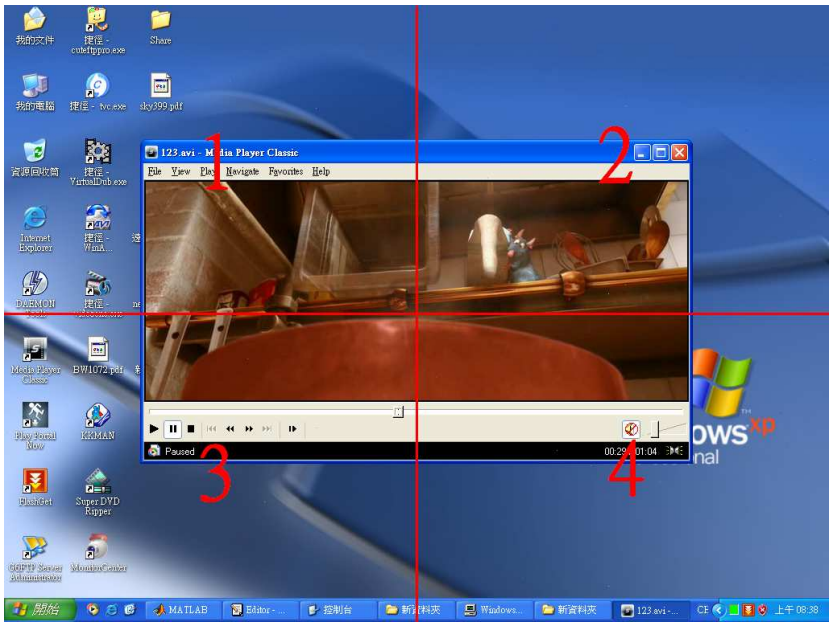


圖 3-25.螢幕四等分

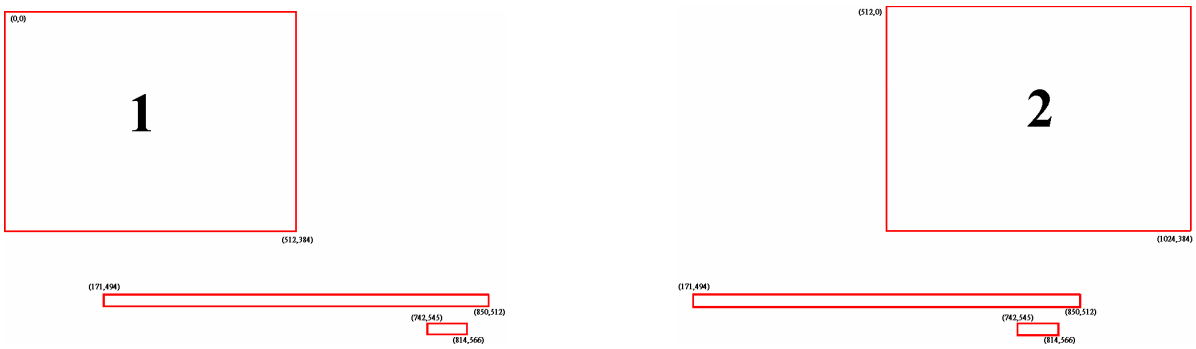


圖 3-26.第一、二次觸發的範圍

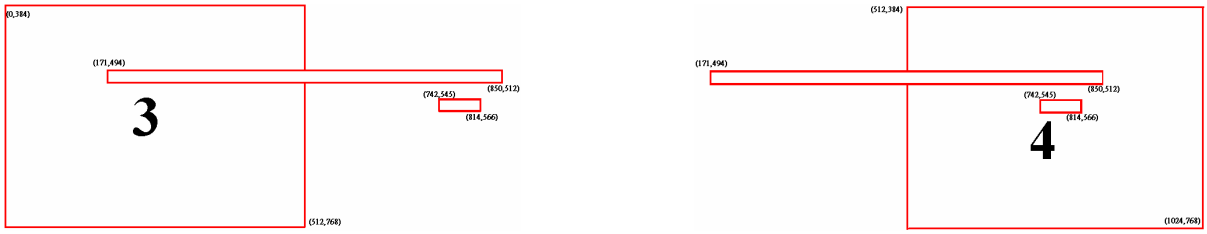


圖 3-27.第三、四次觸發的範圍

●若勾選了 Poll Foreground Window

如果使用者勾選了 Poll Foreground Window 選項，表示他要將目前前景的應用程式視窗範圍加進去。假設延續上個例子，目前的前景程式是 mplayer，Server 便會將整個 mplayer 的視窗大小累加到 rgncache 中，因此整個播放器的區域範圍就是在這樣子加進來的。

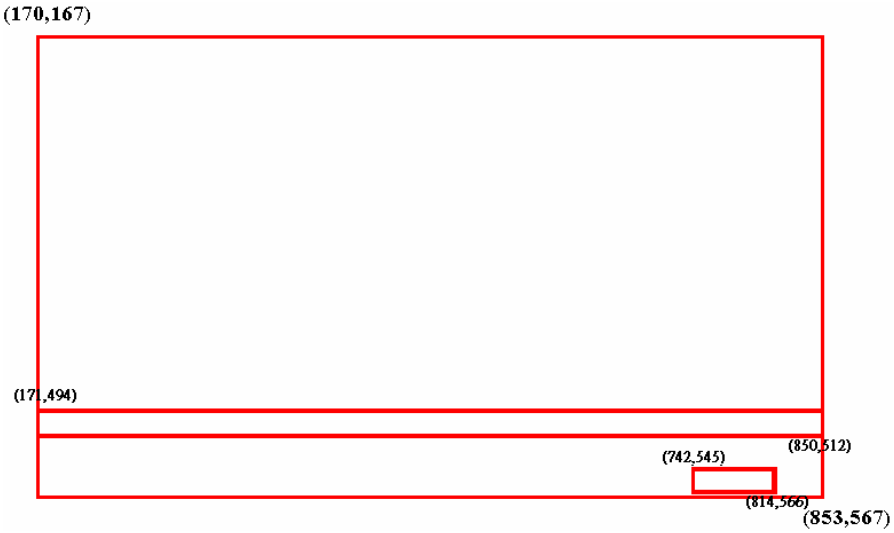


圖 3-28.Poll foreground window 累加範圍

●若勾選了 Poll Window Under Cursor

如果勾選了 Poll Window Under Cursor 後，且游標位置是在後方的小算盤，此時額外加進來的區域則會是小算盤的視窗大小，最後範圍如圖。

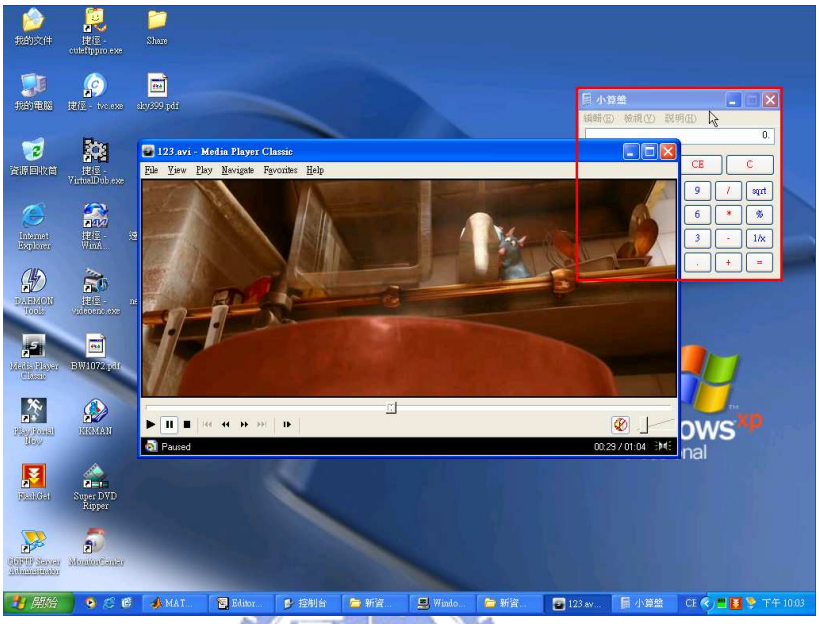


圖 3-29.原始圖

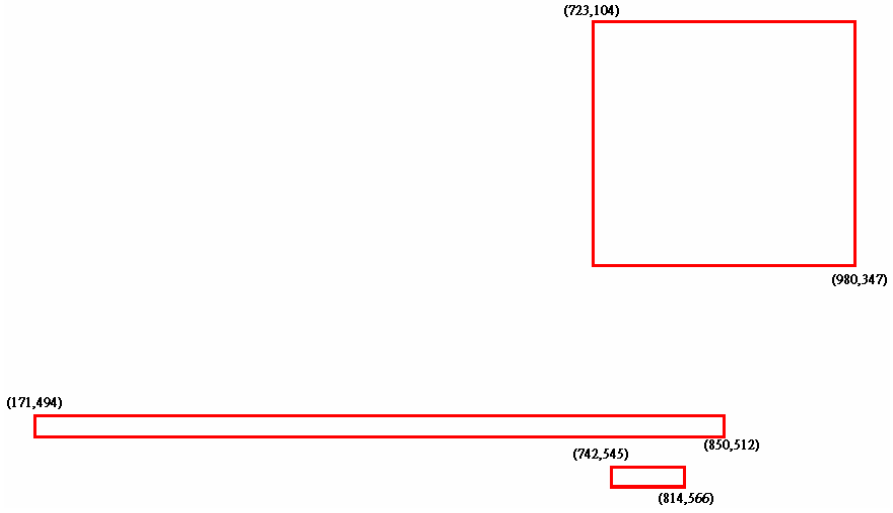


圖 3-30.Poll Window Under Cursor 累加範圍

以上就是當使用者用 mplayer 播放視訊檔案時，每當 Server 清空佇列後，在 Add extra region 一次的累計範圍。獲得上述範圍後，會呼叫 SendUpdate 函數做後續處理。

PS：若同時間有許多 Client 連線進來，則 vncDesktopThread 取得初步範圍後，在 Update Flag、Add extra region、SendUpdate 這些程序，依照幾個 Client 就會重覆幾次，但是否要更新，端由各自的 vncClientThread 是否有將 Update Flag 設為 True，因此，同時有愈多 Client 存在，更新速度可能變慢。

3.4.4 SendUpdate

這個函數是 VNC Server 輸出的部分，它會從桌面的 Framebuffer 上取出，由 TriggerUpdate 函數中獲得的可能變動區域的最新 Pixel 資料，將這些資料更新到 mainbuff，並和 backbuff 做比對，再將真正有變動的部分記錄下來，最後壓縮這些真正變動區塊資料並傳送給 Client。下圖是此函數的流程圖。接著我們會循序漸進的說明這些細節部分。

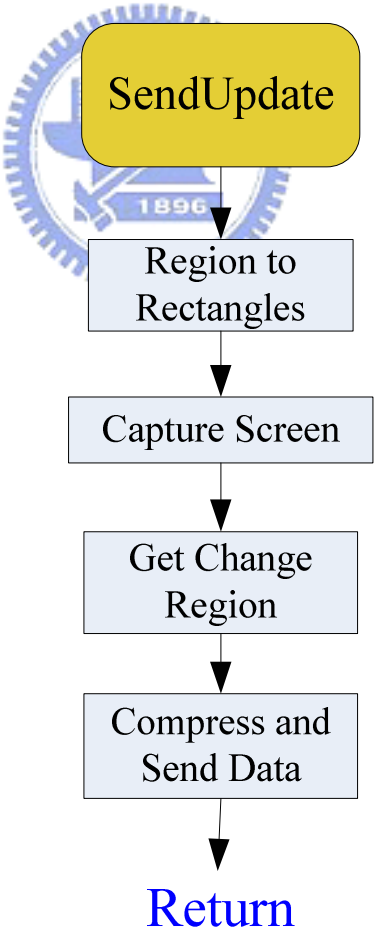


圖 3-31.SendUpdate 流程圖

●Region to Rectangles

在 TriggerUpdate 中，rgncache 記錄著最後可能的變動區域，而 rgncache 是 Windows 的一種類別：Region，它能夠記錄不規則的形狀區域，因此，我們可以在 vncDesktopThread 先記錄 Hook 判斷的可能範圍，再記錄 TriggerUpdate 中額外的區域！那這不規則的區域又要如何使用呢？

```
DWORD GetRegionData(
   HRGN hRgn,                  // Region 的名稱
   DWORD dwCount,              // size of region data buffer
   LPRGNDATA lpRgnData         // region data buffer
);
```

GetRegionData 是 Windows 的 SDK 函數，它可以把不規則的 Region 切成一連串的矩形結構，放入 lpRgnData 中，因此它承載著組成此 Region 的所有矩形，因此只要將他們一個一個取出來處理即可。假設有由 TrigguerUpdate 獲得的區域為 Region 1，如下圖，當使用了 GetRegionData 函數，它便會將此 Region 以橫軸為主，分割成許多個小矩形，將這些矩形的結構以類似陣列的方式，置放在 lpRgnData 中，如圖 30。此即為組成此 region 的矩形來源，稍後會陸續用到。

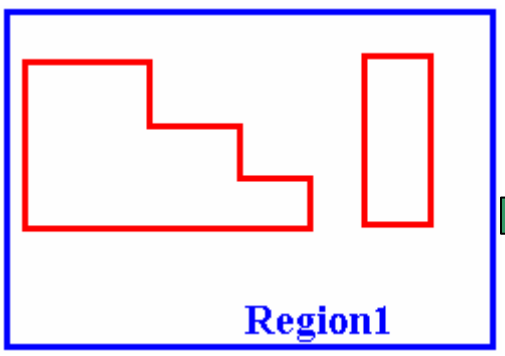


圖 3-32.Region1 示意圖

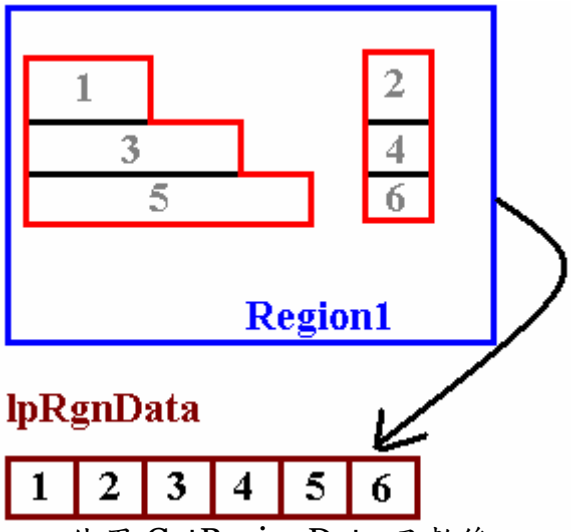


圖 3-33.使用 GetRegionData 函數後

●Capture Screen

在 vncDesktopThread 初始化中的 EnableOptimisedBlits()函數，就有提到，為了讓 mainbuff 可以迅速獲取桌面 Framebuffer 最新資料，那時候有設定 DIB Section，則此時只需要兩個步驟，mainbuff 就可得到最新的 Pixel 資料。

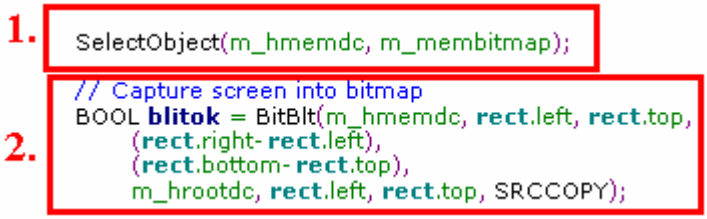


圖 3-34.Capture Screen

1. 將 DIB Section 放入一開始建好的 m_hmemdc 中，讓他們同步。
2. 從螢幕桌面的 Device Context 直接用 BitBlt 函數，從 m_hrootdc 複製所有可能變動的矩形位置資料到 m_hmemdc 中，此時 DIB Section 也會跟著更新，而 mainbuff 的指標位置就是指向對應 DIB Section 的 bitmap 陣列，由此串連起來，每跑一次，mainbuff 的資料就會更新成目前桌面最新的畫面。

BitBlt 函數是用來傳送一個區塊的位元資料（bit-block transfer），意即使在記憶體內的位元圖，傳送到目的端的 Device Context。

```
BOOL BitBlt(
HDC   hdcDest,      // 目的地 DC 的 handle
int   nXDest,       // 目的地 DC 的左上 x 座標
int   nYDest,       // 目的地 DC 的左上 y 座標
int   nWidth,       // 傳送資料區塊的寬
int   nHeight,      // 傳送資料區塊的高
HDC   hdcSrc,       // 來源 DC 的 handle
int   nXSrc,        // 來源地 DC 的左上 x 座標
int   nYSrc,        // 來源地 DC 的左上 y 座標
DWORD dwRop         // 操作的參數
);
```

●Get Change Region

將 mainbuff 的資料更新成最新的狀態後，接著要進行最後的比對，因為整個可能變動區域中，有些是為了避免誤判額外加入的，但實際卻沒有變動到，如果沒有變動範圍的資料也處理的話，會造成傳輸到 Client 的量太大且無意義，因此，這個地方最大的目的是找出真正有變動的部分，並只對這些部分做壓縮和傳輸。

在 vncDesktopThread 迴圈中，我們獲得了最後變動區域，由 GetRegionData 可以得到這個區域矩形陣列。在 Capture Screen 函數中，將桌面對應矩形最新畫面更新到 mainbuff 中，但 backbuff 仍然是舊的畫面資料，因此在這兩個 buffer 中比對這些矩形區域，只要有不相同的地方，就代表是真正發生變動的區域。因為 mainbuff 和 backbuff 都是記憶體，且兩個 buff 都有指標可以指向他們的記憶體位置，因此，只要知道畫面每個像素幾位元和兩個 buffer 各自的指標位置，便可算出對應每個矩形的記憶體起始位置。一開始會先將矩形切成許多 32*32 pixel 的區塊，旁邊不足 32 的用餘數當大小。從左至右，從上至下，以 32*1 pixel 為單位(邊邊不足 32 的區塊以餘數*1 為單位)，比較 mainbuff 和 backbuff 對應的區塊，當有不同的 32*1 出現時，將不同的地方記錄在 toBeSent 的 Region 中，並從 mainbuff 中複製這個 32*1 的資料到 backbuff，這樣的動作會一個矩形接著一個矩形，完成後，toBeSent 中便會記錄著最後真正有變動的部分，且 mainbuff 和 backbuff 內的資料會更新成一模一樣，直到下一次的比對。

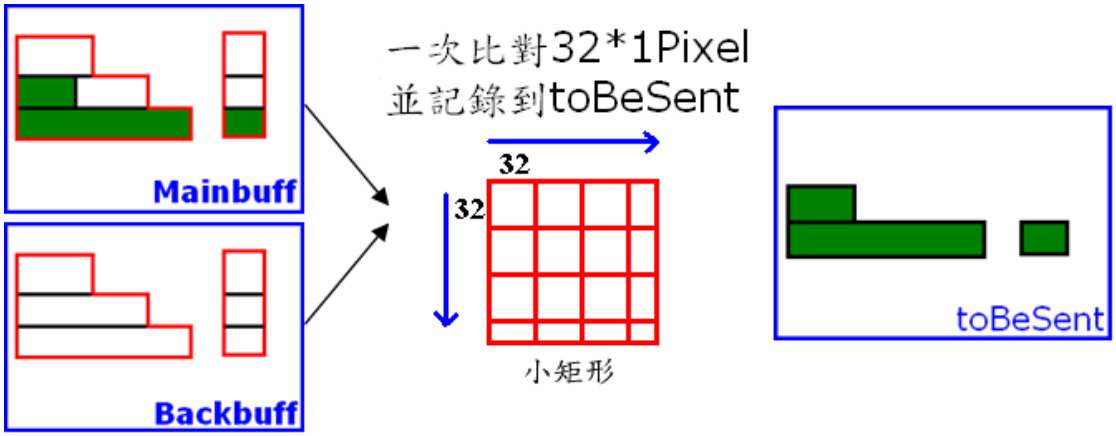


圖 3-35.toBeSent

●Compress and Send Data

在我們取得真正變動區域，並記錄在 toBeSent 中，接著我們也是使用 GetRegionData()將組成 toBeSent 的所有矩形取出，並對他們做處理，如下圖。

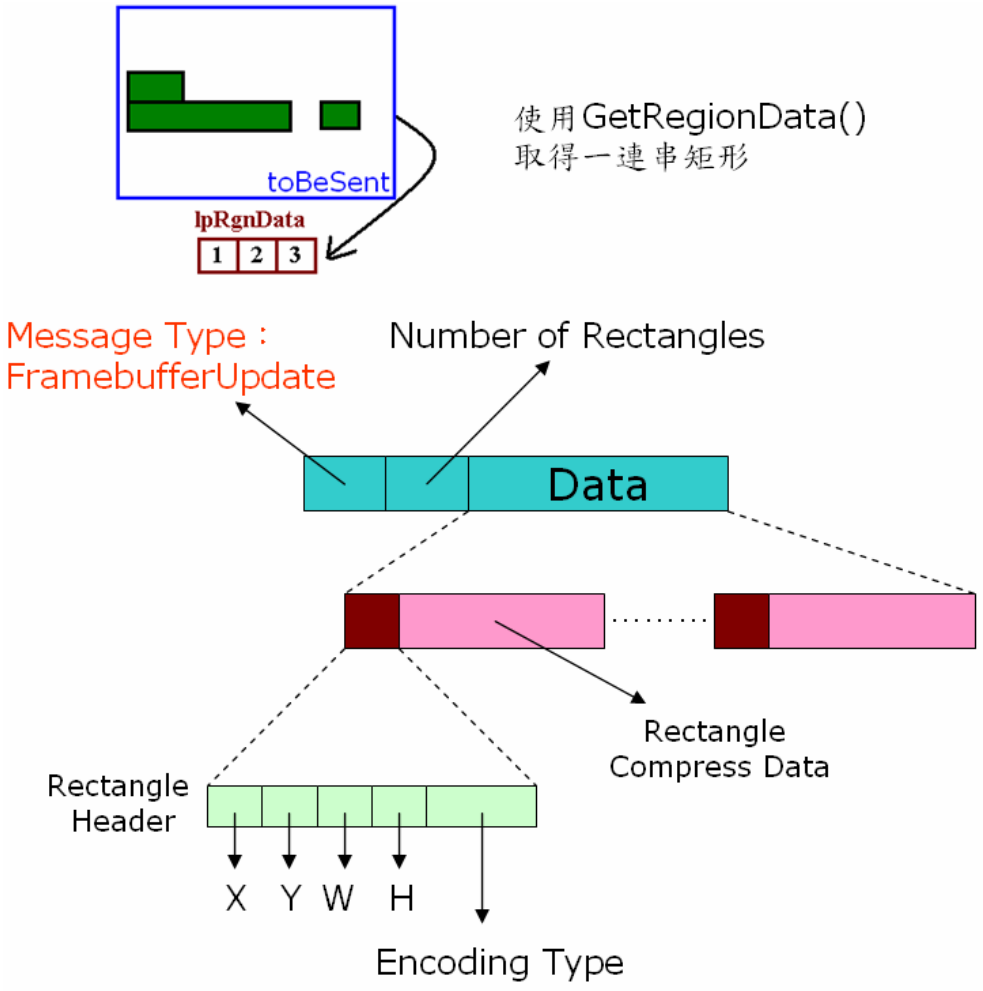


圖 3-36.FramebufferUpdate 訊息

取得變動矩形後，我們要將這些矩形的資料，以 FramebufferUpdate 的訊息傳送給 Client。訊息第一個參數是 Message Type，說明這是何訊息，接著說明這次更新共有幾個小矩形資料所組成，最後會接著這些矩形的資料。每個矩形資料又分為標頭和真正畫面壓縮資料，標頭記載著這個小矩形的位置座標和大小，讓 Client 可將這個矩形資料更新至正確的位置。

第四章 VNC Client

在討論過 VNC Server 的完整運作流程後，應該對遠端桌面系統的 Server 端有了一定的概念，緊接著要介紹對應的 Client 端。Client 所需要的功能是：由滑鼠、鍵盤直接操控 Server，並接收 Server 所傳送過來的畫面資料顯示在 Viewer 程式視窗上，因此，此章要說明它是如何偵測到鍵盤、滑鼠的動作並通知 Server，和如何傳送畫面新更要求、接收 Server 的畫面資料。此章也是以 VNC Client 為架構，詳細說明遠端桌面 Client 的成形。在 4.1 節中簡單介紹 Client 和畫面更新流程；4.2 節詳細說明 VNC Client 的程式建構程序，如一開始建立 Socket、Server 和 Client 溝通版本、安全性、建立 Client Viewer 的工作視窗、設定 Encoding，到後續的接收 Server 每種訊息；4.3 節是整章重點，針對三大部分詳細說明：一、Client 得到 Server 畫面更新訊息後，解碼後如何透過 GDI 將資料畫在 Viewer 視窗程式上；二、說明兩種情況會觸發畫面更新要求：1.Client 接收完畫面更新訊息並顯示在螢幕上之後觸發，和 2.Viewer 最小化還原後觸發；三、得到 OS 發送給 Client Viewer 視窗的鍵盤、滑鼠訊息後，Client 會將這些訊息通知 Server。

4.1 VNC Client 介紹

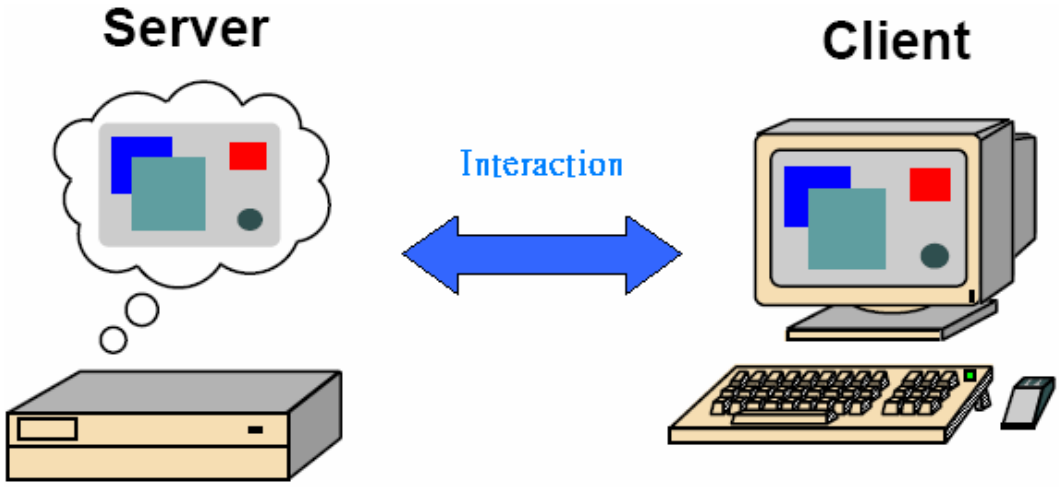


圖 4-1.Server and Client

VNC Client 是使用者的操控端，它只需要擁有基本輸出入的電腦即可，可以單純對 Server 發送一些基本的要求訊息：如敲擊鍵盤、移動、點擊滑鼠、或是要求畫面更新…等，再來就是解析處理 Server 送來的訊息，並將 Server 的畫面顯示在 Client 端，一直持續傳送要求訊息和接收回應訊息。後面章節中會詳細介紹這些動作。

●Client Pull

VNC 整個系統是 Client Pull，即是由 Client 主導，當 Client 要求更新時，Server 才能送資料過來，用下圖來解釋，當 Client 送出 rfbFramebufferUpdate 訊息時，Server 會將 Update Flag 設為 1，而 vncDesktopThread 是否傳送的依據即是由 Update Flag 決定，在此旗標為 1 才能夠傳送。若 Client 沒有要求，此旗標會等於 0，如果此時 vncDesktopThread 有偵測到變動後要進行更新時，會因為旗標為 0，Thread 無法進行更新，因此它便會將此變動範圍先記錄下來，等待下次的更新。

●簡易 Client 畫面更新流程

下面流程是先針對 Client 要求畫面更新部分進行初步講解。在 Server 和 Client 兩者皆完成了必要的初始化程序後，Client 會先發出畫面更新要求的訊息給 Server，Server 端的 vncClientThread 收到訊息後，會由訊息的標頭判斷是何格式，當訊息是 rfbFramebufferUpdateRequest 時，vncClient Thread 會馬上將 Update Flag 設為 1，用來通知 vncDesktopThread，Client 端要求畫面更新，請 vncDesktopthread 將偵測到畫面變動的區域進行編碼，並傳送編碼後的資料給 Client。

Client 端在收到 Server 送過來的訊息，由標頭得知是 rfbFramebuffer-Update，依照其中的編碼格式進行解碼，並將得到的資料在畫面上進行更新，如此 Client 便可得到 Server 目前 Framebuffer 的畫面，當 Client 處理完訊息後，

會再回去觸發一次 rfbFramebufferUpdateRequest 給 Server，接著回去等待 Server 傳送過來的訊息。

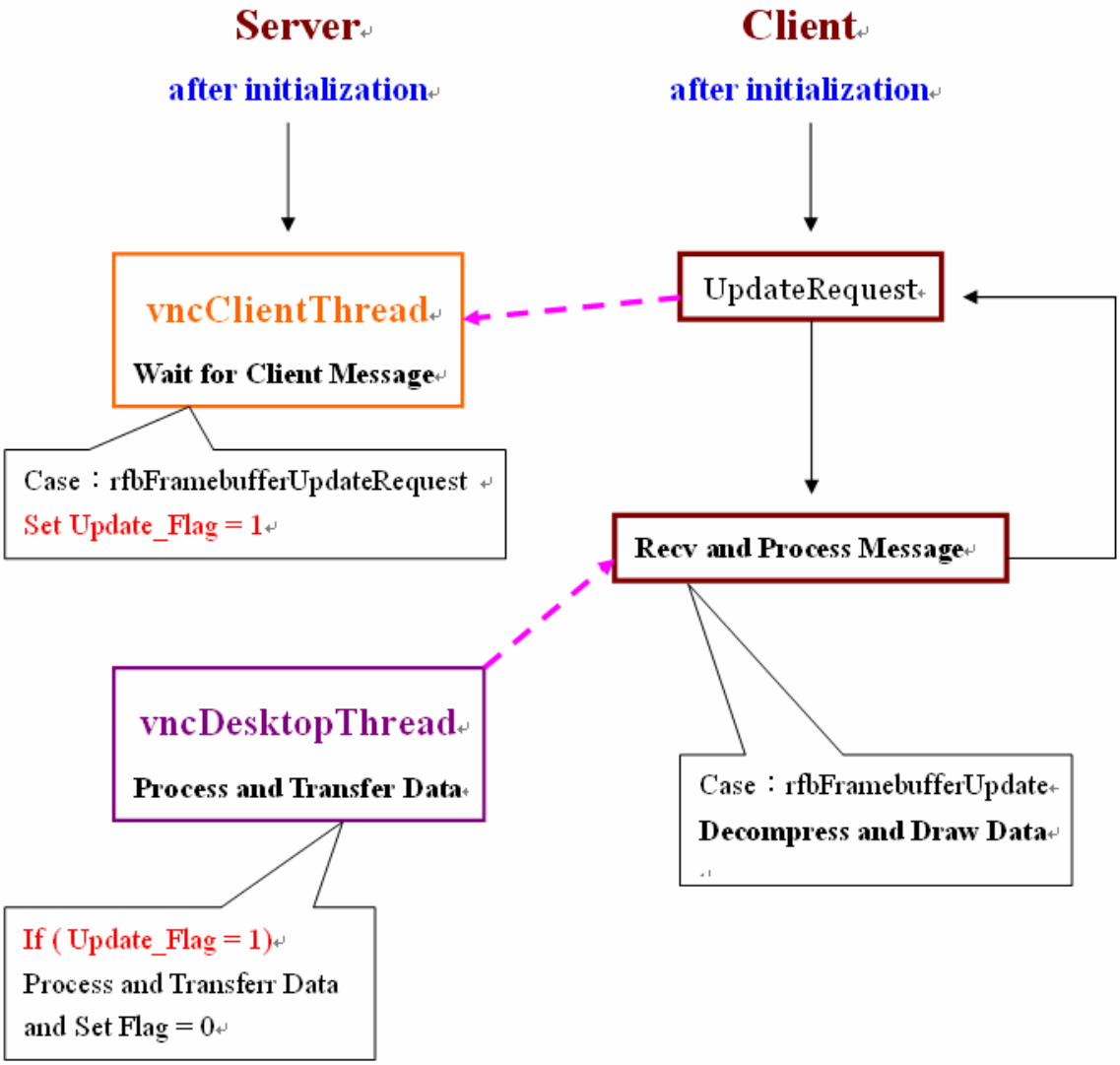


圖 4-2.Client Screen Update Request

Server 傳出編碼後畫面資料的同時，也會將 Update Flag 設為 0，否則 vncDesktopThread 會不停地送封包過去，造成 Client 端處理出現問題。因為 Client 在解碼訊息和處理畫面更新需要時間，即使不斷地傳送封包過去，可能因為 Client 端處理速度不夠快而無力處理，因此，只有在 Client 處理完後，才會再送更新訊息要求將 Flag 設為 1，等待下一次的畫面更新。

4.2 VNC Client 運作流程

這節要詳細說明 VNC Client 完整的成形，從開始的一連串初始化(密碼確定、RFB 版本溝通、Framebuffer 與 Viewer 視窗的建立、到設定 Encoding 和之後解碼格式)，接著初始化完成後，為了取得 Server 初步畫面，Client 會先要求全螢幕的 Framebuffer 更新，以上完成後，隨即進入一無窮迴圈，持續接收 Server 的訊息，如畫面更新、設定調色盤、Bell、Server 剪貼簿的文字、RFB 延伸訊息…等，並做出他們對應的處理。下圖是整個 Client 完整程序的流程圖

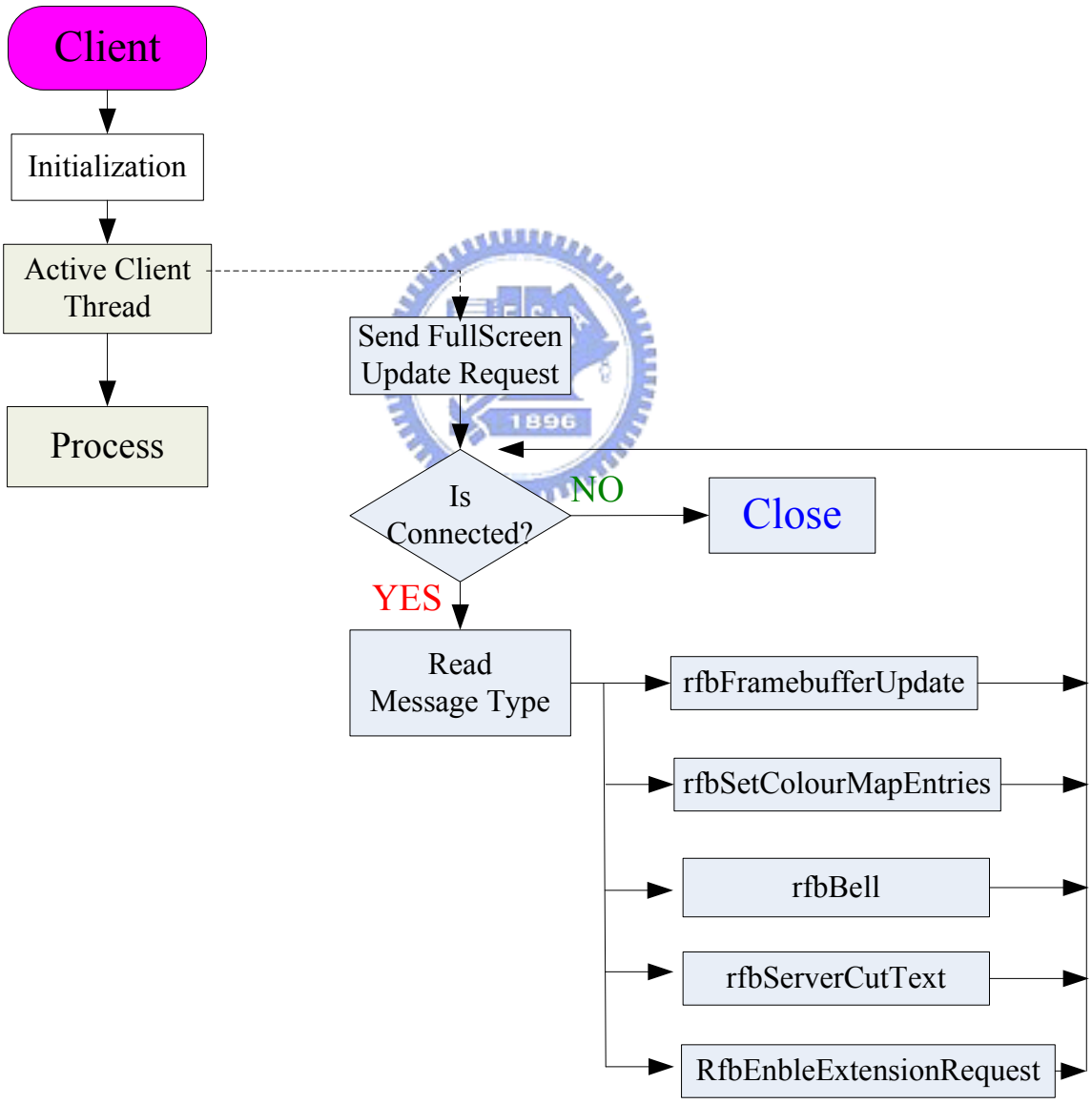


圖 4-3.Client 流程圖

4.2.1 Initialization

這小節所做的設定，都是為了後續運作所需，如建立和 Server 溝通的 Socket、雙方認定的 RFB Protocol、密碼認証、建立 Client 端的 Framebuffer 和 Viewer 視窗、互相傳送初始化訊息、設定 Encoding 和之後解封包會用到的許多參數，這一連串運作後續會一一解釋說明。

```
//--Initialization-------------------
    // Connect if we're not already connected
    if (m_sock == INVALID_SOCKET)
        Connect();

    SetSocketOptions();

    NegotiateProtocolVersion();

    Authenticate();

    CreateDisplay();   // Set up windows etc

    SendClientInit();

    ReadServerInit();

    CreateLocalFramebuffer();

    SetupPixelFormat();

    SetFormatAndEncodings();
//-----------------------------------
```

圖 4-4.Initialization

●Connect()

此函數中會建立 Client 端的 TCP/IP 設定，並和 Server 做三方交握，之後取得 Connect_Socket_id，後續要傳送和接收任何封包皆是透過此 Socket。

●SetSocketOptions()

設定 TCP/IP 的一些功能，這部分不是那麼重要，就不細談。

● NegotiateProtocolVersion()

一開始會由 Server 傳送 Protocol Version 給 Client，讓 Client 知道 Server 最高可以支援的 RFB Protocol 版本，而 Client 會接收此訊息並做判斷，之後回傳訊息告知實際是用哪一個 protocol！為了使雙方都可以向下支援，它不會要求高於 Server 的 protocol version。這部分在第五章有更詳細的說明。

● Authenticate()

在 Protocol Version 被決定後，Server 必須和 Client 溝通連線安全性的種類。這部分在第五章有更詳細的說明。

●CreateDisplay()

因為 VNC Client 需要有一個視窗介面程式：VNCviewer，讓使用者看到 Server 的畫面並透過它操控 Server，因此，這邊即是建立一個視窗程式。在第二章的 Win32 程式建立流程中，已初步講解它所需的幾個關鍵區塊，而這個 CreateDisplay()函數，即是其中最重要 InitWindow()的細節。

```
// Create the window
WNDCLASS wndclass;

wndclass.style          = 0;
wndclass.lpfnWndProc    = ClientConnection::WndProc;
wndclass.cbClsExtra     = 0;
wndclass.cbWndExtra     = 0;
wndclass.hInstance      = m_pApp->m_instance;
wndclass.hIcon          = LoadIcon(m_pApp->m_instance, MAKEINTRESOURCE(IDI_MAINICON));
wndclass.hCursor        = LoadCursor(m_pApp->m_instance, MAKEINTRESOURCE(IDC_DOTCURSOR));
wndclass.hbrBackground  = (HBRUSH) GetStockObject(BLACK_BRUSH);
wndclass.lpszMenuName   = (const TCHAR *) NULL;
wndclass.lpszClassName  = VWR_WND_CLASS_NAME;

RegisterClass(&wndclass);

m_hwnd = CreateWindow(VWR_WND_CLASS_NAME,
                      _T("VNCviewer"),
                      winstyle,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,
                      CW_USEDEFAULT,     // x-size
                      CW_USEDEFAULT,     // y-size
                      NULL,              // Parent handle
                      NULL,              // Menu handle
                      m_pApp->m_instance,
                      NULL);
```

圖 4-5.CreateDisplay

上圖中是 Win32 視窗程式建立的過程，第一個紅框框宣告 WINDCLASS 的基本設定，裡面較重要的是藍色框框，它定義了 Viewer 對應的訊息處理函數 WndProc，即所有發送給 Viewer 視窗的訊息，都會在這個函數中做處理，因此，當使用者對此 Viewer 視窗敲擊鍵或滑鼠，OS 都會以訊息發送給 WndProc，它收到後會將這些訊息轉成 RFB Protocol 定義的封包格式傳送給 Server，如何轉換的這部分在 4.3 節中會完整說明。

宣告並設定 WNDCLASS 後，要透過 RegisterClass 將它註冊到 Windows 中，將著要用 CreateWindow()函數將此 Viewer 程式建立起來，第一個參數即是上面的 WINDCLASS name，第二個參數是 Viewer 視窗程式的名稱：“VNCviewer“，後面的是它的 x、y 座標和高、寬度。最重要的是，CreateWindow()函數的回傳值，m_hwnd，它是此 Viewer 視窗的 HWND，後續 OS 都是由這個 HWND 來和 Viewer 視窗打交道。到此，一個 Viewer 程式已建立完成，但因為目前沒有任何 Server 的畫面資料，因此還不能將把 Viewer 程式秀在螢幕上。

●SendClientInit()
●ReadServerInit()

在 Server 溝通好版本、安全性種類、認證正確，即會進入初始化程序，Client 會先送 ClientInit 訊息給 Server，通知是否准許有其他 Client 同時存在。等 Server 收到設定完，會回傳 ServerInit 訊息，來知會 Client 目前 Server 的 Framebuffer 狀況！！這部分在第五章會再詳細說明。

●CreateLocalFramebuffer()

```
TempDC hdc(m_hwnd);
m_hBitmap = ::CreateCompatibleBitmap(hdc,
     m_si.framebufferWidth,
     m_si.framebufferHeight);

// Select this bitmap into the DC with an appropriate palette
ObjectSelector b(m_hBitmapDC, m_hBitmap);
PaletteSelector p(m_hBitmapDC, m_hPalette);

// Put a "please wait" message up initially
RECT rect;
SetRect(&rect, 0,0, m_si.framebufferWidth, m_si.framebufferHeight);
COLORREF bgcol = RGB(0xcc, 0xcc, 0xcc);
FillSolidRect(&rect, bgcol);

COLORREF oldbgcol  = SetBkColor(  m_hBitmapDC, bgcol);
COLORREF oldtxtcol = SetTextColor(m_hBitmapDC, RGB(0,0,64));
rect.right = m_si.framebufferWidth / 2;
rect.bottom = m_si.framebufferHeight / 2;

DrawText (m_hBitmapDC, _T("Please wait - initial screen loading"), -1, &rect,
     DT_SINGLELINE | DT_CENTER | DT_VCENTER);
SetBkColor(  m_hBitmapDC, oldbgcol);
SetTextColor(m_hBitmapDC, oldtxtcol);

InvalidateRect(m_hwnd, NULL, FALSE);
```

圖 4-6.CreateLocalFramebuffer()

接收完 ServerInit 訊息後，我們就取得 Server 的桌面狀態，因此在這個函數，就要將 Viewer 視窗初步外框秀出。函數一開始第一個紅框框中，會建立和 Server 桌面相同大小的 Bitmap 記憶體，這邊建立 Bitmap 記憶體也是為了之後能夠快速更新畫面，因為後續只要將畫面像素資料複製到此塊記憶體，再透過 BitBlt 函數即可更新變動區塊到螢幕。

第二塊紅框框是因為尚未取得 Server 的初始畫面，在此會使用 GDI 的函式，先將 Viewer 視窗的工作區間背景設為灰色，並在要上面輸出“Please wait - initial screen loading“這串字。第三塊紅框框 InvalidateRect ()觸發後，OS 會偵測到無效區域產生，它會傳送 WM_PAINT 給 Viewer 的訊息處理函數 WndProc()，裡面會將剛剛設定的畫面秀在螢幕，如下圖。外面框框的名稱是 Server 的電腦名稱，背景是灰色，在左上角的位置印有 Please wait - initial screen loading “。

圖 4-7.Initial Viewer

●SetupPixelFormat()

這邊是設定 ServerInit 訊息中的 PIXEL_FORMAT 參數，將這些參數記錄下來，在後續工作會用到。

●SetFormat And Encodings()

傳送 Client 目前螢幕的 PIXEL_FORMAT 給 Server，並將內部支援的編碼格式，依使用者的優先順序排好，以 rfbSetEncodings 訊息傳送給 Server。

以上就是所有的 Initialization，這一連串的工作都是為了後續運作正常，接著下一節要說明初始化後，Client 會接收 Server 送來哪些訊息。

4.2.2 Client 訊息處理迴圈

一連串初始化過程後，進入迴圈前，Client 會先送一次畫面更新要求訊息，要求 Server 立即送全螢幕的資料過來，接著，Client 開始專心處理 Server 送過來的訊息。至於詳細處理封包格式部分，在第五章會再詳細說明。

表 4-1.Server 傳給 Client 的訊息種類

Number	Name
0	**FramebufferUpdate**
1	**SetColourMapEntries**
2	**Bell**
3	**ServerCutText**

表 4-2.其他有定義的訊息種類

Number	Name
255	**Anthony Liguori**
254,127	**VMWare**
253	**gii**

要使用非以上定義的訊息，Server 須收到 Client 回傳的確認訊息才可以.

4.3 VNC Viewer 細部工作原理

這節是整章最重要的部分，因為一套遠端桌面系統的 Client 端，不外乎是送滑鼠、鍵盤和要求畫面更新訊息給 Server，接著就是接收 Server 送過來的溝通或畫面訊息，並將他們處理後顯示畫面在視窗上，因此，這節就是要針對這些動作做分析。在 4.3.1 先說明 Client 收到 Server 畫面更新訊息解碼後，怎麼用 GDI 函式將這些像素顯示在視窗上，4.3.2 接著說明三種情況會觸發畫面更新要求訊息：1.在 Client 畫面更新後會觸發，2.在 Viewer 視窗縮小還原後觸發，3.一開始的觸發全螢幕更新，4.3.3 節則是針對 Client 如何將本地端鍵盤和滑鼠的動作，轉成 RFB Protocol 訊息的方式傳送給 Server。以上說明後，就可完全了解 Client 一切的運作。

4.3.1 Viewer 畫面更新

要更新一個完整 FramebufferUpdate 訊息中的畫面，一共需要三個步驟，分別是：1.解碼、2.更新像素資料至 Viewer 的 Framebuffer、3. InvalidateRect() 和 WM_PAINT，先由 FramebufferUpdate 得知有幾個矩形資料，再各自呼叫解碼函數將畫面資料解出，並更新到 Viewer 的 Framebuffer 上，再用 InvalidateRect()函數將對應矩形位置無效化，此時 OS 會發送 WM_PAINT 給 Viewer 的訊息處理函數，在處理函數中將新的畫面資料更新到 Viewer 視窗上，以上步驟一直不斷重覆，使用者即可獲得 Server 的最新畫面。

●1. 解碼

在 Client 訊息處理迴圈中，當它判斷收到 FramebufferUpdate 訊息後，會呼叫 ReadScreenUpdate()函數進行處理，一開始先讀出此更新訊息中共有幾個矩形資料，接著用 for 迴圈一個一個再取出，如下圖藍框。每個矩形都有自己的標頭，裡頭記錄著這個矩形它 x、y 座標、高、寬和編碼格式，因此，依照不同的

編碼格式呼叫不同的解碼函數將畫面資料解出，如下圖第一個紅框。

●2.更新像素資料至 Viewer 的 Framebuffer

在 Initialization 時，Client 已經建立了一塊 Bitmap 記憶體，用來存放 Client 的 Framebuffer，因此每個壓縮的矩形資料解碼後，會將解完的像素複製到這塊記憶體上，重覆以上動作就可達到 Client 端 Framebuffer 更新。但至此，Viewer 視窗上的內容還不會更新，之後要再透過 BitBlt 函數將這些矩形部分的畫面，從 Bitmap 記憶體複製到螢幕，Client 的 Viewer 視窗才能看到變動後的畫面，這些部分是以 InvalidateRect()來觸發 WM_PAINT 訊息，並在 Viewer 的視窗處理函數中完成，接著會繼續說明。

```
void ClientConnection::ReadScreenUpdate() {
    rfbFramebufferUpdateMsg sut;
    ReadExact((char *) &sut, sz_rfbFramebufferUpdateMsg);
    sut.nRects = Swap16IfLE(sut.nRects);

    for (UINT i=0; i < sut.nRects; i++) {
        rfbFramebufferUpdateRectHeader surh;
        ReadExact((char *) &surh, sz_rfbFramebufferUpdateRectHeader);
        surh.r.x = Swap16IfLE(surh.r.x);
        surh.r.y = Swap16IfLE(surh.r.y);
        surh.r.w = Swap16IfLE(surh.r.w);
        surh.r.h = Swap16IfLE(surh.r.h);
        surh.encoding = Swap32IfLE(surh.encoding);

        switch (surh.encoding) {
        case rfbEncodingRaw:
            ReadRawRect(&surh);
            break;
        case rfbEncodingCopyRect:
            ReadCopyRect(&surh);
            break;
        case rfbEncodingRRE:
            ReadRRERect(&surh);
            break;
        case rfbEncodingCoRRE:
            ReadCoRRERect(&surh);
            break;
        case rfbEncodingHextile:
            ReadHextileRect(&surh);
            break;
        case rfbEncodingJPEG:       //Todd
            DecompressJpegRect(surh.r.x, surh.r.y, surh.r.w, surh.r.h);
            break;
        default:
            break;
        } ? end switch surh.encoding ?
        RECT rect;
        rect.left   = surh.r.x - m_hScrollPos;
        rect.top    = surh.r.y - m_vScrollPos;
        rect.right  = rect.left + surh.r.w;
        rect.bottom = rect.top  + surh.r.h;
        InvalidateRect(m_hwnd, &rect, FALSE);
    } ? end for UINT i=0;i<sut.nRects;i++ ?

    PostMessage(m_hwnd, WM_REGIONUPDATED, NULL, NULL);

} ? end ReadScreenUpdate ?
```

圖 4-8.Viewer 畫面更新

●3.InvalidateRect()和 WM_PAINT

在解碼完一個小矩形畫面後，Bitmap 記憶體中對應區塊，已經記錄著最新的畫面資料，接著要將這些區塊的像素更新到螢幕上，就要透過 GDI 的 BitBlt 函數來達成。

```
SelectObject(m_hBitmapDC, m_hBitmap);
BitBlt(hdc, ps.rcPaint.left, ps.rcPaint.top,
ps.rcPaint.right- ps.rcPaint.left, ps.rcPaint.bottom- ps.rcPaint.top,
m_hBitmapDC, ps.rcPaint.left+m_hScrollPos, ps.rcPaint.top+m_vScrollPos, SRCCOPY);
```

圖 4-9.Copy to Screen

VNC Client 會先透過 InvalidateRect()來將 Viewer 對應的矩形位置區域無效化，當有無效區域產生後，OS 會偵測到 Viewer 的某些地方需要更新重畫，OS 隨即發送 WM_PAINT 給 Viewer 視窗，請它對應的視窗處理函數 WndProc()處理，當 WndProc 判斷是 WM_PAINT 訊息時，它便會從此訊息得到需要更新的座標和區域大小，接著使用 BitBlt 函數，將 Bitmap 記憶體對應區塊的資料複製到螢幕，程式碼如上圖。先用 SelectObject()將 Bitmap 記憶體同步至 m_hBitmapDC，再呼叫 BitBlt 函數把來源 m_hBitmapDC 的資料複製到螢幕的 Device Context，處理完成後，Viewer 視窗的矩形位置就會更新成最新畫面。

以上就是畫面如何更新。假設一個畫面更新訊息中有 5 個矩形的壓縮資料，就要重覆以上動作 5 次才會完成一次的更新！

4.3.2 Viewer 觸發畫面更新要求

接著這節要介紹，Client 在以下三種情況下會觸發畫面更新要求，分別是：1. 一次完整畫面更新後；2.在 Viewer 視窗縮小還原後；3.一開始的全螢幕更新。

●1. 在一次完整畫面更新後

延用上圖，在一次完整的更新後，即所有矩形資料皆解碼後，會跳離 for 迴圈，在 ReadScreenUpdate 的尾端(橘色框)，Client 以訊息的方式 Post 一個

WM_REGIONUPDATED 訊息給 Viewer 視窗程式，它所屬訊息處理函數 WndProc 偵測到 WM_REGIONUPDATE 訊息之後，會先看 Viewer 狀態有否被使用者縮小(下圖程式碼)，這是因為 VNC 是 Client Pull，當判斷目前 Viewer 被縮小(m_dormant=1 時)了，表示 Client 暫時不需更新畫面，它就不會要求畫面更新，如下圖紅框。如果 Viewer 沒有被縮小(m_dormant=0)才會發送一個要求畫面更新的訊息給 Server，亦即，在一個完整的更新完成後，且視窗沒有被縮小的情況下，Client 才會再傳送一次畫面更新要求訊息。

```
if (!m_dormant){
    SendIncrementalFramebufferUpdateRequest();
}
```

圖 4-10.觸發畫面更新訊息

在 Client 的解碼過程中需要時間，因此 Server 都會記錄著在這段時間所發生的變動，當 Client 上一次畫面訊息處理完成並傳送要求畫面更新訊息後，Server 才會將 Client 解碼那段時間，所發生的變動畫面資料一起壓縮送過來，因此，一直重覆這些動作，Client 的 Viewer 視窗就能持續更新成 Server 最新的畫面。

●2.在 Viewer 視窗最小化還原後

在 Viewer 被最小化或還原時，OS 各自會發送 SC_MINIMISE、SC_RESTORE 的訊息，給 Viewer 的訊息處理函數，當 WndProc()收到後會判斷是何子息，如果是 SC_MINIMISE，它會馬上呼叫 SetDormant()函數，將 m_dormant 狀態會被設為 TRUE，代表現在 Client 被最小化，狀態相當於靜止，則 Client 不會主動要求畫面更新；當視窗被使用者還原時，WndProc()會收到 SC_RESTORE 的訊息，它馬上會呼叫 SetDormant()函數，會將 m_dormant 設為 FALSE，而且因為 Viewer 視窗時被最小化的期間不會發送要求畫面更新訊息，因此目前 Client 的 Framebuffer 是停留在 Server 最後一次更新時候的狀態，在 Viewer 視窗還原後需要馬上傳送一個畫面更新要求，請 Server 將最後一次更新之後所有變動的畫面區域資料，用一個更新訊息全部送過來。

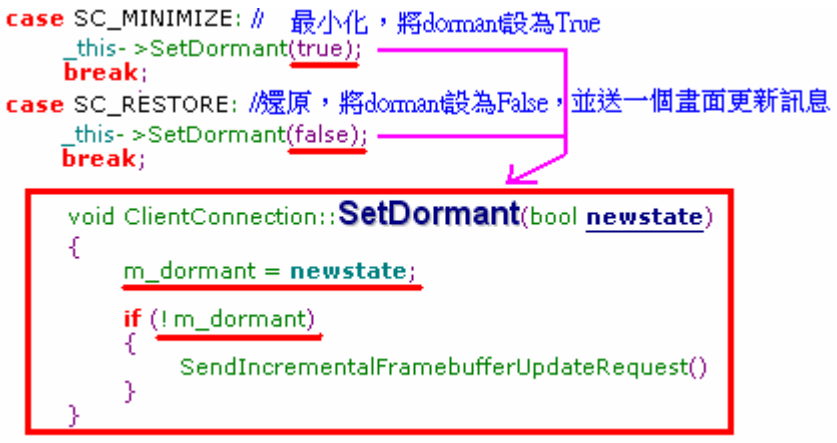


圖 4-11.SetDormant()函數

●3 一開始的全螢幕更新

這個情況只會發生一次，即 Client 初始化完成後，因為 Client 的 Framebuffer 尚未有任何畫面資料，因此它需要請 Server 傳送完整的全螢幕畫面資料。

以上就是 Client 可能要求畫面更新的三種機制。

4.3.3 鍵盤、滑鼠動作訊息

這一小節要說明 Viewer 如何將滑鼠和鍵盤的訊息傳送給 Server。下面是 Viewer 視窗對應的訊息處理程式 WndProc 中，當使用者敲擊滑鼠和鍵盤時，OS 都會以訊息的方式傳送到 Viewer 視窗的訊息佇列，而視窗程式的訊息迴圈會從佇列中取出訊息，呼叫 WndProc()訊息處理函數來對滑鼠或鍵盤敲擊訊息加以處理。

●滑鼠動作訊息

藍色框框中有對應滑鼠按下左鍵、中鍵、右鍵和移動所產生的訊息種類，當以上動作產生時，OS 會發送訊息給 Viewer，此時 WndProc 會對他們做處理，紅色框框是由傳過來的副消息取出滑鼠位置，再呼叫 ProcessPointerEvent()函數

將這些訊息，轉成 Server 看得懂的 RFB Protocol 封包格式傳出去。內部細節只是一些鍵值轉換，在這地方就不細部說明。

```
case WM_LBUTTONDOWN:
case WM_LBUTTONUP:
case WM_MBUTTONDOWN:
case WM_MBUTTONUP:
case WM_RBUTTONDOWN:
case WM_RBUTTONUP:
case WM_MOUSEMOVE:
    {
        if (!_this->m_running) return 0;
        if (GetFocus() != hwnd) return 0;
        if ( _this->m_opts.m_ViewOnly) return 0;

        int x = LOWORD(lParam);
        int y = HIWORD(lParam);
        _this->ProcessPointerEvent(x,y, wParam, iMsg);

        return 0;
    }
```

圖 4-12.Mouse Event

下圖則是敲擊鍵盤時訊息處理，和滑鼠的相似，OS 也會發送訊息給 WndProc()函數，它則是呼叫 ProcessKeyEvent()函數來轉換他們的鍵值，將這些鍵值轉成符合 RFB Protocol 格式傳送給 Server。

```
case WM_KEYDOWN:
case WM_KEYUP:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
    {
        if (!_this->m_running) return 0;
        if ( _this->m_opts.m_ViewOnly) return 0;

        _this->ProcessKeyEvent((int) wParam, (DWORD) lParam);

        return 0;
    }
```

圖 4-13.KeyEvent

透過 Viewer 視窗的 WndProc()訊息處理函數，將鍵盤滑鼠對 Viewer 動作訊的息轉成 Server 能夠認知的格式，在 Server 模擬出來，再由 Server 將變動畫面傳送資料給 Client，如此循環不止，透過以上複雜的機制，一套完整的遠端桌面程式即可完成。

第五章 Server 和 Client 的溝通訊息

此章統整所有在三、四章中討論過的訊息，將他們細步的封包格式，和 Server、Client 接收後如何處理，在這一節詳細說明。本章依照 RFB Protocol 的三個 phase，Handshaking Phase、Initialization Phase、Interaction Phase，將所有訊息歸類成三類，他們各自會在這三個 phase 中出現。5.1 節說明 Handshaking Phase 中，溝通 RFB Protocol Version、安全性種類、密碼認證的動作和訊息；5.2 節說明在 Initialization Phase 中，初始化的 ClientInit 和 ServerInit 訊息；5.3 節分為兩部分，一、Client 送給 Server 的訊息；二、Server 送給 Client 的訊息；5.4 節則是介紹 VNC 的四種內建的壓縮法的方法：RAW、CopyRect、RRE、Hextile、ZRLE，和他們的各自的封包格式。

5.1 Handshaking Phase

這些各在 Server 的 vncClientThread Initialization 和 Client 的 Initialization 中完成，在這個 phase，主要是溝通雙方的 RFB Protocol 版本、安全性種類和密碼認証，這些完成後，代表 Server 准許 Client 進行連線，才會接著後面的 Initialization Phase。

●NegotiateProtocolVersion

一開始會由 Server 傳送 Protocol Version 給 Client，讓 Client 知道 Server 最高可以支援的 RFB Protocol 版本，而 Client 收到後會回傳訊息，告知實際是用哪一個 protocol！但它不能夠要求高於 Server 的 protocol version。這個機制是為了使雙方都可以向下支援。目前發布的 RFB 版本有 3.3，3.7，3.8。新增進的編碼方法和格式，不會改變 protocol 的版本，因為 Server 可以忽略它不懂得的編碼。

Protocol Version 訊息包含了 12 個 bytes，用一連串的 ASCII 字元解譯成“RFB xxx .yyy \n”，xxx 和 yyy 是版本前面的字和後面的數字。

表 5-1.Protocol Version

No.of Bytes	Value
12	“RFB 003.003\n”(hex 52 46 42 20 30 30 33 2e 30 30 33 0a)
12	“RFB 003.007\n”(hex 52 46 42 20 30 30 33 2e 30 30 37 0a)
12	“RFB 003.008\n”(hex 52 46 42 20 30 30 33 2e 30 30 38 0a)

●Authentication

在 Protocol Version 被決定後，Server 必須和 Client 溝通連線安全性的種類。RFB 版本 3.7 或以上 Server 會先列出它所支援的安全性種類：

表 5-2.Security Message

No.of bytes	Type	Description
1	U8	number-of-security-types
number-of-security-types	U8 array	security-types

表 5-3.Security-types：

Number	Name
0	Invalid
1	None
2	NVC Authentication
5	RA2
6	RA2ne
16	Tight
17	Ultra
18	TLS
19	VeNCrpt

表 5-4.回傳錯誤訊息格式

No. of bytes	Type	Description
4	U32	reason-length
reason-length	U8 array	reason-string

Server 會列出至少一個支援的有效安全性種類，Client 收到後回傳一個 byte，表明此連線將使用哪個安全性種類！如果出現問題時，如：LoopBack、Server 未設密碼、或 number-of- security-types= 0 時，這個連線會中斷。之後 Server

會傳送一個 ASCII 的字串描述理由，並在傳回錯誤訊息後馬上關閉連線！

因為 VNC 大部分版本都一定有 None、VNC Authentication 這兩種，因此僅對此兩個稍做介紹：

1. None：不須認証，且 protocol 的資料以不加密的方式傳送。
2. VNCAuthentication：使用 VNC 認証，protocol 以不加密的方式傳送。Server 會先送一個隨機 16-byte 的 challenge。Client 收到 challenge 後會以 DES 來解碼，解完後回傳一個 16-byte 的 response。Server 收到 response 會比對是否正確。

在 Security 交握後，Server 會回傳一個 4bytes 的 Security Result 給 Client，告知 security handshaking 是否成功。不成功，在 3.8 版會傳錯誤訊息告知原因，而 3.3 和 3.7 版則是直接中斷連線！

表 5-5.change and response

No. of bytes	Type	Description
16	U8	challenge response

表 5-6.Security Result

No. of bytes	Type	Description
4	U32	Status
	0	OK
	1	Failed

最後，Server 會回傳訊息，告知實際是用哪一個 protocol！雙方接者進行下一步的初始化過程！

5.2 Initialization Phase

在 Server 溝通好版本、安全性種類、認證正確，即會進入初始化程序，這部分是在 Server vncClientThread 的 Initialization 後半部分，和 Client Initalizaton 中間過程。Client 會先送 ClientInit 訊息給 Server，告知是否准許同時有不同的 Client 和 Server 連線。Server 收到 ClientInit 後設定完，會回傳 ServerInit 訊息，來知會 Client 目前 Server 的 Framebuffer 狀況！！這部分說明兩個訊息詳細的封包和格式。

●ClientInit 訊息

表 5-7.ClientInit 訊息

No.of bytes	Type	Description
1	U8	Shared-flag

當 Shared-flag 非零值(True)，Server 准許其他 Client 同時連進 Server。

而 Shared-flag 為零值(False)，則同一時間只准許一位 Client 使用！

●ServerInit 訊息

Server 收到 ClientInit 的訊息後，會回送 ServerInit 訊息通知 Client，Server 的 framebuffer 的寬、高、PIXEL_FORMAT、和遠端桌面的名字。

表 5-8.ServerInit 訊息

No.of bytes	Type	Description
2	U16	Framebuffer-width
2	U16	Framebuffer-height
16	**PIXEL_FORMAT**	Server-pixel-format
4	U32	name-length
name-length	U8 array	name-string

PIXEL_FORMAT 是用來辨識 Server 的 pixel 格式，它會一直被使用，直到 Client 用 SetPixelFormat 訊息來更動它！

表 5-9.PIXEL_FORMAT

No.of bytes	Type	Description
1	U8	btis-per-pixel
1	U8	depth
1	U8	big-endian-flag
1	U8	true-colour-flag
2	U16	red-max
2	U16	green-max
2	U16	blue-max
1	U8	red-shift
1	U8	green-shift
1	U8	blue-shift
3		padding

◎**Bits-Per-Pixel**：一個 pixel 值是用幾個 bits 來表示，目前必須是 8、16、或 32，而少於 8bits 的目前還未支援。

◎**Big-Endian-Flag**：多 byte pixel 值的排序方式，當然這對 8 bits-per-pixel 是無意義的。

◎**True-Colour-Flag 為非零值(True)**：則接續後面六項是用來解出 pixel 中，紅綠藍顏色的強度，red-max 是紅色的最大值(= $2^n - 1$，n 代表紅色佔了幾個 bit)，它是以 big-endian 表示。Red-shift 表示從最大的位元算起，須位移幾個 bit 來取得紅色的值。green- max, green-shift, blue-max, blue-shift 同理。

舉例，從一個給定的 pixel 值，取出紅色的成分：

1. 依照 big-endian-flag 和本地的系統，轉成和系統相符合的
2. 向右移 red-shift 個 bits，找到紅色值的位置
3. 由 red-max 可知，紅色的值總共有幾個 bits

如此即可得到紅色的強度值。

◎**True-Colour-Flag 為零值(False)**：則 Server 傳過來的 Pixel 值，不是由紅綠藍的顏色所組成，而是由 colour map 為索引，由索引值對應出真正的 Pixel 值。而 colour map 是 Server 用 SetColour MapEntries 訊息來設定。

5.3 Interaction Phase

Handshaking Phase、Initialization Phase 結束後，隨即會進入 Client 和 Server 兩邊的訊息溝通，至於如何處理的部分，Server 是在 vncClientThread 初始化完成後會進入一訊息處理迴圈，而 Client 則是 Viewer 程式建立程序完成後，也會馬上進入訊息處理迴圈，在兩邊溝通的過程中，會有許許多多的訊息封包格式，裡面各自蘊含了什麼功能和意義，在這節中會詳細介紹。此節共會分為兩小節，5.3.1 是討論 Client 送給 Server 的訊息；5.3.2 節則是討論 Server 送給 Client 的訊息。如果要傳送非以上訊息，需要確定兩端都支援，否則 VNC 會忽略甚至終止。

5.3.1 Client 送給 Server 的訊息

下表是 Protocol 中已訂定 Client 送給 Server 的訊息。

表 5-10.Client to Server messages

Number	Message Type
0	**SetPixelFormat**
2	**SetEncodings**
3	**FramebufferUpdateRequest**
4	**KeyEvent**
5	**PointerEvent**
6	**ClientCutText**

1. SetPixelFormat

此訊息是用來設定會在 FramebufferUpdate 訊息中用到的 pixel 格式值。如果 Client 端沒有送出 SetPixelFormat 的訊息，Server 就一直依照初始化時，在 ServerInit 訊息中所設定的格式來編排 pixel 值！如果 true-colour- flag 是零值 (False)，則表示 colour map 要被使用。即使 Server 之前有設定過對應表，只要 Client 送出 colour map 是空的，Server 必須立刻用 SetColourMapEntries 來設定 colour map 的對應表(entry)。

表 5-11.SetPixelFormat

No.of bytes	Type	[Value]	Description
1	U8	0	message-type
3			padding
16	**PIXEL_FORMAT**		pixel-format

表 5-12.PIXEL_FORMAT

No. of bytes	Type	Description
1	U8	bits-per-pixel
1	U8	depth
1	U8	big-endian-flag
1	U8	true-colour-flag
2	U16	red-max
2	U16	green-max
2	U16	blue-max
1	U8	red-shift
1	U8	green-shift
1	U8	blue-shift
3		Padding

2. SetEncodings：

設定編碼種類，Server 會依照所設定的編碼來壓縮 pixel 資料。Client 一開始會將預設偏好的編碼種類，照次序放在這個訊息裡傳給 Server (愈前面的愈優先)，而 Server 可以採納，也可以不理會這個提示。如果沒有明確定義編碼方式，則 Server 會以 raw 的方式編碼 pixel 資料。除了原本的編碼，Client 可要求" pseudo-encodings" ，告訴 Server 它支援某種 protocol 的延伸。若 Server 不支援，它會直接忽略。如果 Client 沒收到 Server 對 psudo-encoding 的確認，就知道 Server 不支援。

表 5-13.SetEncodings

No.of bytes	Type [Value]	Description
1	U8 2	Message-type
1		padding
2	U16	number-of-encodings

後面會接續 number-of-encodings 個編碼種類

表 5-14.Encoding-Type

No.of bytes	Type [Value]	Description
4	S32	encoding-type

3. FramebufferUpdateRequest：

用來告訴 Server，Client 想要 framebuffer 中的某一塊區塊，並以 x、y 座標、長和寬來表示此面積！Server 會用 FramebufferUpdate 來回覆。

表 5-15.FramebufferUpdateRequest

No. of bytes	Type [Value]	Description
1	U8 3	message-type
1	U8	incremental
2	U16	x-position
2	U16	y-position
2	U16	width
2	U16	height

Server 會假設 Client 保留 Framebuffer 的所有部分，所以一般來說，Server 只需要傳送有變動的區塊給 Client。但是，可能發生某些原因(如第一次的全螢幕更新)，Client 失去了 Framebuffer 的資料，則在發出 FramebufferUpdateRequest 訊息時，裡面的 incremental 就要設為零(False)，Server 看到後會送出整個完整的 Framebuffer 資料，而且不會使用 CopyRect encoding 傳送。如果 Client 沒有喪失面積中任何的資料，則 incremental 會設為非零值(True)，

Server 在收到 FramebufferUpdateRequest 訊息後，vncClientThread 會馬上將 Update Flag 設為 TRUE，只有當畫面的某些區塊產生變動時，Server 才會送出這些區塊的畫面資料。也因此，當 Client 發出 Request 後，如果 Server 部分都沒發生變動，那麼 FramebufferUpdate 和 Request 之間可能會持續無限長的時間。

4. KeyEvent：

一個按鍵的按下或釋放。按鍵按下時 Down-flag 為非零值(True)，按鍵釋放 down-flage 為零值(False)。

表 5-16.KeyEvent

No.of bytes	Type [Value]	Description
1	U8 4	message-type
1	U8	down-flag
2		padding
4	U32	key

表 5-17.一些常用的鍵

Key name	Keysym value
BackSpace	0xff08
Tab	0xff09
Return or Enter	0xff0d
Escape	0xff1b
Insert	0xff63
Delete	0xffff
Home	0xff50
End	0xff57
Page Up	0xff55
Page Down	0xff56
Left	0xff51
Up	0xff52
Right	0xff53
Down	0xff54

Key name	Keysym value
F1	0xffbe
F2	0xffbf
F3	0xffc0
F4	0xffc1
…..	….
F12	0xffc9
Shift(left)	0xffe1
Shift(right)	0xffe2
Control(left)	0xffe3
Control(right)	0xffe4
Meta(left)	0xffe7
Meta(right)	0xffe8
Alt(left)	0xffe9
Alt(rigth)	0xffea

5. PointerEvent：

表示游標的移動或按下。目前游漂的位置(x-y 的座標)和 button-mask 的 0~7 個 bits 中，分別表示 1~8 個狀態，0 表示上，1 表示按下。在傳統的滑鼠，按鍵 1、2、和 3 分別代表左鍵、中間和右鍵。而有滾輪的話，向上滾代表按鍵 4 的按下又放開，向下滾代表按鍵 5 的按下又放開。

表 5-18.PointerEvent

No. of bytes	Type [Value]	Description
1	U8 5	message-type
1	U8	button-mask
2	U16	x-position
2	U16	y-position

6. ClientCutText：

表示 Client 在它的剪貼簿中有新的東西。在行的結尾用新行(ASCII 的 0AH)代表，不需要歸位(ASCII 的 0DH)。不能傳 Latin-1 的字元以外的字。

表 5-19. ClientCutText

No. of bytes	Type [Value]	Description
1	U8 6	message-type
3		padding
4	U32	length
length	U8 array	text

PS:換行符號是以兩個字元表示，一般而言，第一個字元是 ASCII 字元的 0DH，它表示歸位（carriage return）的意思，即游標回到此行的最左邊。第二個字元是 0AH，它表示 line feed 的意思，也就是到下一行（假如有下一行的話）或增加一行（假如沒有下一行，也就是此行是最後一行的話）的意思。

5.3.2 Server 送給 Client 的訊息

下表是 Protocol 中已訂定 Client 送給 Server 的訊息。

表 5-20.Server 傳給 Client 的訊息種類

Number	Name
0	**FramebufferUpdate**
1	**SetColourMapEntries**
2	**Bell**
3	**ServerCutText**

表 5-21.其他有定義的訊息種類

Number	Name
255	**Anthony Liguori**
254,127	**VMWare**
253	**gii**

要使用非以上定義的訊息，Server 須收到 Client 回傳的確認訊息才可以.

1. FramebufferUpdate

它是 Server 用來回覆 FramebufferUpdateRequest。Framebufferupdate 包含了一連串矩形畫面的壓縮資料，Client 收到後，會將他們一一解碼，並存進 Client 建立的 Framebuffer(初始化中的 Bitmap 記憶體)中。

表 5-22.Framebufferupdate

No.of bytes	Type [Value]	Description
1	U8 0	message-type
1		padding
2	U16	Number-of-rectangles

後面跟著 number-of-rectangles 個矩形的 pixel 資料，每個矩形包含

表 5-23.每個矩形標頭資料

No.of bytes	Type [Value]	Description
2	U16	x-poisition
2	U16	y-position
2	U16	width
2	U16	height
4	S32	encoding-type

接著會再接著每個矩形畫面壓縮編碼後的資料。

2. SetColourMapEntries：

當 Client 在送出 PIXEL_FORMAT 中的 True-Colour-Flag 為零值，即代表要使用調色盤，Server 隨即會送出這個 SetColourMapEntries 訊息來通知 Client，傳過來的 Pixel 值應該對應到哪個 RGB 的強度，即所謂的調色盤。

表 5-24.SetColourMapEntries

No. of bytes	Type [Value]	Description
1	U8 1	message-type
1		padding
2	U16	first-colour
2	U16	number-of-colours

後面會跟著 number-of-colours 個 colour 資料

表 5-25.對應 Colour 的 RGB 值

No. of bytes	Type Value]	Description
2	U16	red
2	U16	green
2	U16	blue

3. Bell：

如果 Client 有設定 Bell 的話，則會響一聲或是產生對應動作。

表 5-26.Bell

No.of bytes	Type [Value]	Description
1	U8 2	Message-type

4. ServerCutText：

表示 Server 在它的剪貼簿中有新的東西。在行的結尾用換行(ASCII 的 0AH)代表，不需要歸位(ASCII 的 0DH)。不能傳 Latin-1 的字元以外的字。

表 5-27.ServerCutText

No.of bytes	Type [Value]	Description
1	U8 6	message-type
3		padding
4	U32	length
length	U8 array	text

5. rfbEnableExtensionRequest：

這個延伸訊息是給 Client 和 Server 都有設定時用的，但其他版本的 VNC 可能不支援，因此雙方都要確認是否有溝通好格式。

5.4 VNC 內建壓縮方法和資料封包

Server 會以 Framebuffer update 訊息，來傳送畫面更新，訊息前面會記載著這個它承載了幾個矩形，和一連串矩形的標頭和編碼資料，這一小節，就是要來說明解開這些矩形的標頭檔後，後續編碼資料的封包和格式，這些壓縮方法分別是 RAW、CopyRect、RRE、Hextile、和 ZRLE。

表 5-28.壓縮種類

Number	Name
0	Raw
1	CopyRect
2	RRE
5	Hextile
16	ZRLE
-223	DesktopSize pseudo-encoding

除了上面這些外，可能有其他特殊的編碼，但需 Server 和 Client 同時支援才可使用，在此就不做詳細討論。

在圖形傳輸時，根據不同的網路頻寬、Client/Server 處理資料的速度，VNC Protocol 提供了多種編碼方式，主要有 Raw、Copy-Rect、RRE、Hextile 以及 ZRLE，接下來說明這幾種編碼方式：

1. RAW：

最簡單的編碼方式。資料包含了寬*高的 pixel 值，而這些值代表每一行從左到右，由上到下，一直掃描下來的 pixel。所有 Client 都要能夠處理這個編碼，而 Server 在沒有指定編碼方式時，也只會用 raw 的方式傳。

表 5-29.RAW

No. of bytes	Type [Value]	Description
width*height*bytesPerPixel	PIXEL array	pixels

2. CopyRect encoding：

將 framebuffer 中的圖形位置由一塊區域複製到另一塊區域，是一種簡單有效率的方法，Client 端的 buffer 已經暫存了這塊區域的資料，Server 只要將 x、y 的座標告知 Client，複製 buffer 中的資料，便可以在畫面中顯示出來，可以節省網路頻寬。這樣的情形發生在移動桌面內的視窗時，由於視窗內的資料並沒有改變，只需要改變顯示的區域，透過這樣的方法可以降低頻寬的使用。

表 5-30.CopyRect

No.of bytes	Type [Value]	Description
2	U16	src-x-position
2	U16	src-y-position

3. RRE：

這種編碼方式是將同樣的 pixel 壓縮成單一數值重複計算，在 VNC 中實現的方法就是將要傳送的矩型區域中的顏色資訊，找出分布最多的顏色當做背景色，然後根據其他顏色的分布定義出多個子區域（Sub-rectangle)並記錄該區域的顏

色、位置以及大小，如下圖 33。而 Client 只須畫一個矩形，填滿背景 pixel 值，然後把其它的矩形對應到其位置上，即可復原！此方法用在面積大顏色單純的區塊（例如：單色的桌布畫面），但不適合於背景複雜的區塊

表 5-31.RRE

No.of bytes	Type Value]	Description
4	U32	Number-of-subrectangles
bytesPerPixel	PIXEL	Background-pixel-value

後面接著 Number-of subrectangles 個子矩形結構

表 5-32.RRE 的子結構

No. of bytes	Type [Value]	Description
bytesPerPixel	PIXEL	Subrect-pixel-value
2	U16	x-position
2	U16	y-position
2	U16	width
2	U16	height

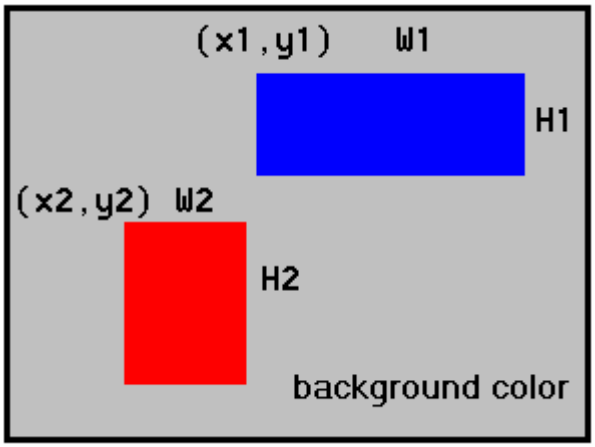


圖 5-1.RRE

4. Hextile：

Hextil 是 RRE 的一種變形。一個矩形區域以 tile 為單位，由左至右、由上至下依序切割，每個 tile 為 16 x 16 pixels，如果一個矩形區域，寬不是 16 的倍數那麼最後一個 tile 的高度就會窄一點，相同情形，如果高不能被 16 整除，那麼最後一個 tile 的寬度就會短一點。每個 tile 可以使用 Raw 編碼或是 RRE 編碼。

每個 tile 有自己的背景顏色，如果其背景顏色和前一個 tile 的背景顏色一樣，就不需要明確定義；另如果所有子矩形的 pixel 值都一樣，則整個 tile 只需要記錄前景 pixel 一次，和背景 pixel 類似，前景 pixel 也可以從前面的 tile 取得。

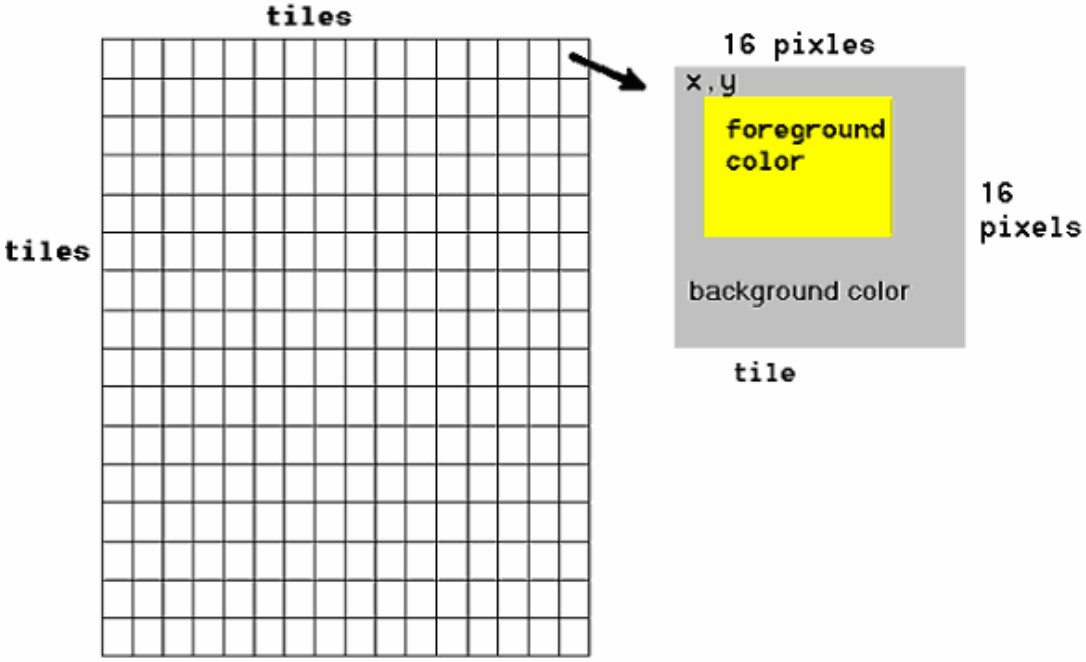


圖 5-2.Hextile 編碼

所有資料是由照順序編碼的 tile 所組成的，每個 tile 都由 subencoding type byte 開始，也就是一些 bits 組成的遮罩。

表 5-33.subencoding 遮罩

No.of bytes	Type [Value]	Description
1	U8	subencoding-mask:
	1	Raw
	2	BackgroundSpecified
	4	ForegroundSpecified
	8	AnySubrects
	16	SubrectsColoured

Raw:如果 Raw bit 被設定了,則其他的 bits 都無效;後面會跟著 width*height 個 pixel 值(width 和 height 是 tile 的寬和高)

BackgroundSpecified：如果被設定了，後面會跟著這個 tile 的背景 pixel 值

如果這個 bit 沒有被設定，那背景 pixel 值和目前最後面的 tile 一樣。但當出現第一個不是用 raw 編碼的 tile，這個 bit 就必須被設定。

表 5-34.BackgroundSpecified

No.of bytes	Type Value]	Description
bytesPerPixel	PIXEL	background-pixel-value

ForegroundSpecified：如果被設定了，後面會跟著這個 tile 所有子矩形的前景 pixel 顏色。這個 bit 若被設定，則 SubrectsColoured bit 必須設為 0。

表 5-35.ForegroundSpecified

No.of bytes	Type [Value]	Description
bytesPerPixel	PIXEL	foreground-pixel-value

AnySubrects：若設定的話，會跟著一個 byte，記錄著子矩形的數量沒有設定的話，就沒有任何子矩形(換言之，整個 tile 只有背景的顏色)

表 5-36.AnySubrects

No.of bytes	Type [Value]	Description
1	U8	number-of-subrectangles

SubrectsColoured：AnySubrects 設定的話，須處理每個子矩形的 pixel 值。子矩形的定義如下：

表 5-37.SubrectsColoured

No.of bytes	Type [Value]	Description
bytesPerPixel	PIXEL	subrect-pixel-value
1	U8	x-and-y-position
1	U8	width-and-height

位置和大小，被定在兩個 bytes 內，在 x-and-y-position 的前四個 bits 為 x 座標，後四個 bits 為 y 座標，而 width-and-height 的前四個 bits 為寬減 1，後四個 bits 為高減 1 如果沒有設定的話，所有的子矩形都是同一個顏色(前景的顏色)，如果 ForegroundSpecified bit 沒有設定，那就和最後一個 tile 的前景顏色一樣。

5. ZRLE：

Zib Run-Length Encoding（ZRLE），是結合了 zlib 壓縮、titlng、調色盤和 rub-length 編碼技術。傳輸的時候，矩形區域以 4 個 bytes 開始，緊接著是 zlib 壓縮資料，唯一的 zlib stream 傳送在 RFB protocol 連線上，因此 ZRLE 區域必須精確的依序編碼和解碼。Zlib 資料在還沒解壓縮之前，代表了 64 x 64 個 pixels，由左到右、從上到下，類似於 hextile 切割方式，如果高和寬不是 64 的整數倍，那麼最後一筆資料區域會小一點。

第六章 實做與實驗數據分析

本論文以可實際運作的遠端桌面程式--VNC 為主軸，在第三、四章中詳細討論了 Server 和 Client 各是如何實作出來，和他們如何互相溝通運作。說明過一套完整遠端桌面程式工作原理後，接著這章要探討每個遠端桌面程式都存在的問題：播放視訊檔案時會導致網路傳輸流量暴增。在 6.1 節中先對這個問題詳細論述，並提出改善方法：針對視訊播放區域加以 JPEG 壓縮；6.2 節說明實作過程和初步不錯的結果，但這部分又延伸了新的問題：播放品質低落，因此，此節後半部分又針對這延伸問題深入探討和解決；6.3 節則是將以上得到的數據綜合比較。

6.1 問題論述和改善方法

每一套實際運作的遠端桌面程式，都是取得 Server 螢幕 Framebuffer 的變動畫面，並將這些畫面資料傳送給 Client，而 Client 只要接收這些資料並更新在 Viewer 視窗上，使用者就能夠得知 Server 的狀態並操控它，這在一般操作的使用並無太大問題，因為畫面的變動範圍和頻率不大，所需要處理的資料量有限，但當使用者播放視訊檔案時，此時估且不論 Frame 大小，在每一秒中畫面更新少則 15、6 個 Frame，多則高達 30 個 Frame，可想而知，Server 處理這些頻繁變動的畫面範圍，將耗掉相當可觀的時間和頻寬。

因此，為了避免需要傳輸這麼龐大的資料量，我們勢必要提出一些方法來因應。在 VNC 原本內部就規範了幾個壓縮方法：Hextile、CoRRE、RRE、RAW(請參閱第五章最後)，但實際播放影片(長度 1 分 04 秒、672*272 24 bits、23.976FPS)來測試所得到的結果卻相當不理想，可見得這些壓縮方式，對於像影片這樣劇烈畫面變動的情況下，發揮不了太大作用，測試數據如下表。

表 6-1.VNC 內部壓縮方法流量統計

Encodings	每秒平均流量(Mb)	總流量(Mb)
Hextile	5.57	356.4
CoRRE	4.17	266.6
RRE	8.31	532.0
RAW	3.63	232.6

表中顯示使用了 VNC 內部編碼方式的測試數據，當 Server 端播放了一段影片，它所需的網路頻寬相當驚人，這並不是一般家庭能夠負荷。因此，我們要提出另一種方法來改善這個問題。因為以上 VNC 內部提供的編碼方式，都是不失真壓縮，這些方法在畫面靜態或是變動較輕微的狀況下，並不會有太顯著的差異，一旦持續播放動態影片時，緊跟而來的就是流量的暴增。

因此，為了對這個現象做改善，我們實做了兩大部分：

一、我們加入了失真壓縮：JPEG，因為遠端桌面程式最重要的就是畫面，而 JPEG 對於畫面所做的破壞性壓縮尚在可接受的範圍，而且採用 JPEG 還可以由 Quality 在流量和畫面品質間做取捨。

二、只針對視訊播放的區塊做壓縮，對於其他區塊採用原本 VNC 內部不失真壓縮方式，因為，若在文字的部分做 JPEG 壓縮，可能會造成相當程度的模糊，為了避免這些情況發生，會盡量只針對視訊播放的區塊做 JPEG 壓縮處理！

6.2 實作過程和改進

這節前半部分 5.2.1 節說明如何實作的 6 大步驟，和初步改善成果數據的測量統計，並和 VNC 內建的編碼方式做比較，後半部 5.2.2 節則是對於初部改善所延伸出來的問題：播放品質低落，進行詳細的探討和改進。

6.2.1 實作過程說明

接續在 5.1 節中討論的，此小節要說明如何只針對視訊播放區塊做 JPEG 壓縮，總共分為 6 大部分：1.先說明 Server 如何得知 Client 支援 JPEG 壓縮；2.接著複習 Server 是如何取得組成畫面真正變動範圍的一連矩形資料；3.取得視訊的播放區域；4.判斷要更新的矩形是否落入視訊播放區域，落在視訊播放區域的會以 JPEG 壓縮；5.不屬於播放畫面可能也會以 JPEG 壓縮的特殊情況；6.啟動 JPEG 或以原本 VNC 的壓縮方式傳送畫面資料；最後是改善後實際測試的流量統計。

●1. Server 如何得知 Client 支援 JPEG 壓縮

因為 VNC 一切訊息都是遵照 RFB Protocol，如果要和市面上其他版本的 VNC 相容，就不能隨易加入規定外的訊息格式，除非雙方都支援，否則會有不相容的狀況發生，因此我們要以另外的方式來通知 Server，Client 有能力支援 JPEG 壓縮。在 Client 的 Initialization 過程中，最後一項 SetEncodings()函數，此函數是要用來通知 Client 所選定的編碼，它的訊息格式如下表。

表 6-2.SetEncodings

No.of bytes	Type [Value]	Description
1	U8 2	Message-type
1		padding
2	U16	number-of-encodings

我們是在這個 SetEncodings 的訊息中，將 padding 加以利用，當 padding 的值設為 75 或 50，代表我們要啟動 Server 的 JPEG 壓縮，而且 JPEG 的 Quality 要設為 75 或是 50，Quality 也可以設定為其他值，目前是先以這兩個值說明。

```
   se->type = rfbSetEncodings;
//--------------Todd
//  se->pad = 75;;          //Todd quality = 75
//  se->pad = 50;;          //Todd quality = 50
//--------------------
```

圖 6-1.Client 透過 SetEncodings 訊息通知

當 Server 接收到 SetEncodings 的訊息時，會額外判斷此訊息的 padding，如果值為 75 或是 50，表示 Client 支援 JPEG 壓縮，Server 會馬上將 JPEG 的旗標設為 TRUE，並將 Quality 設定為 75 或 50，後續只要符合我們設定的狀況，就會將畫面資料以 JPEG 壓縮並傳送出去。透過這個小技巧，即使是其他版本的 VNC 連線進入，也不會有不相容的情形發生。

```
//----------------Todd
m_client->Setsign(FALSE);
if(msg.se.pad ==75)
{
    m_client->Setsign(TRUE);
    m_client->Setquality(75);
}
if(msg.se.pad ==50 )
{
    m_client->Setsign(TRUE);
    m_client->Setquality(50);
}
//-----------------
```

圖 6-2.Server 判斷是否啟動 JPEG

●2.取得所有組成真正變動範圍的矩形資料

在 3.4.4 節中，從 vncDesktopThread 取得的最後範圍，在 SendUpdate 函數裡進行 mainbuff 和 backbuff 的比對，當兩塊記憶體裡對應的這些範圍中有不同，會將不相同的範圍記錄到 toBeSent 的 region 中，並由 GetRegionData()得到組合 toBeSent 的所有矩形資料，並存放在 lpRgnData 中，如圖。

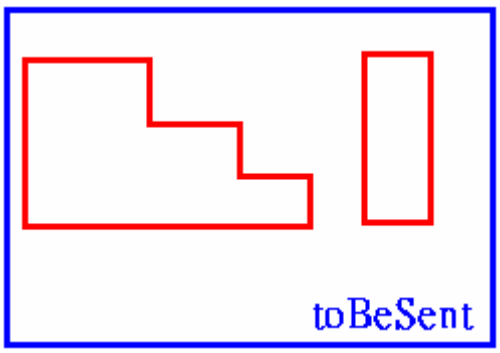


圖 6-3.toBeSent 示意圖

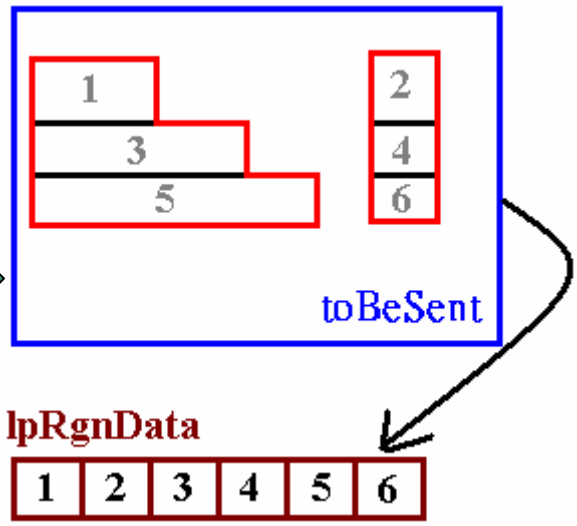


圖 6-4.取得組合 toBeSent 的矩形資料

以上取得的所有矩形畫面區塊，代表都是必須進行壓縮並傳送給 Client 端，因此只需對他們進行一些邏輯判斷，看他們是否有落在視訊播放區內，只要落在其內，都必須以 JPEG 壓縮，若沒有，則採用 Client 原本要求的壓縮方式。假設延用上例，總共有 6 個矩形需要傳送，則下述的步驟就要重複 6 次，因為 Framebuffer update 訊息的標頭會說明總共有幾個矩形，接著才是一個個矩形的壓縮資料，因此，這些動作次數是依照 toBeSent 由幾個矩形組合而定的。

那要如何取得視訊播放區的範圍呢？和如何判斷上面得到的矩形是否落在裡面呢？後續以 Windows Media Player 實際例子來詳細說明！

●3.取得視訊的播放區域

在實際對組成 toBeSent 的矩形進行壓縮前，會先有一連串的邏輯判斷，看是否要啟用 JPEG。一開始先用 Windows 的函式庫 FindWindow()，察看 Window Media Player 是否有被使用者啟用，當它被啟用後，則此函數會回傳 HWND，若沒被啟用則離開。接著再以 GetWindowRect 函數取得 Player 的整個程式區域座標大小，存在 wmplayer_rect 中。以上僅代表 Media Player 程式已被使用者啟動，但如果它被使用者最小化，並沒有顯示在螢幕上，則螢幕上的畫面就不可能有劇烈變動，此時也不需要啟動 JPEG 壓縮，因此，我們要額外用 IsIconic() 函數，看 Media Player 是否被最小化。只有在 Media Player 被啟用且沒被最小化時，wmplayer_sign 才會被設為 TRUE，這就表示播放器目前正被使用中。

```
RECT wmplayer_rect;
BOOL wmplayer_sign = FALSE;
HWND window = FindWindow("WMPlayerApp",NULL)
    if(window !=NULL)
    {
        GetWindowRect(window, &wmplayer_rect)
        if(!IsIconic(window))    //若不是最小化，回傳0
            wmplayer_sign = TRUE;
    }
```

圖 6-5.取得視訊播放區域

●4.判斷要更新的矩形是否落入視訊播放區域

當 wmplayer_sign 被設為 TRUE，代表目前 Media Player 被啟動，並已取得 Media Player 的範圍，存於 wmplayer_rect 的矩形結構，接著要判斷組成 toBeSent 的矩形，是否有落入播放器的區域。這邊只要用 IntersectRect()函數，即可得知兩個矩形是否有交叉到，只要和 Player 交叉到，IntersectRect()會回傳 1，此時就會 jpegrect 的旗標設為 TRUE，表示此矩形資料要以 JPEG 壓縮。

```
RECT interrect;
if(wmplayer_sign)
{
   if(IntersectRect(&interrect,&wmplayer_rect,&rect))
        Setrectsign(TRUE);
}
```

圖 6-6.矩形是否落入視訊播放區域

下圖是矩形交叉的可能，當這 9 種交叉可能發生，則此矩形就屬於視訊變動的範圍內，因此我們就要對這個矩形進行 JPEG 壓縮。

圖 6-7.交叉可能

●5.不屬於播放畫面可能也會以 JPEG 壓縮的特殊情況

因為要精準定位出矩形是否在視訊播放區，可能要花相當多複雜判斷才能辦到，因此採用交叉的方式來判斷，可能會發生有些不屬於視訊部分也會被壓縮的情形。如圖 6-8 左.當使用者還原一個比了 Media Player 還大的視窗，如開啟 Word 或是 Excel 檔案，則變動的矩形會超過播放器範圍，雖然畫面被視窗蓋掉，但此時的矩形也和播放器交叉到，即使不屬於視訊部分仍然會以 JPEG 壓縮。圖 6-8 右也是，當有其他視窗部分和播放器重疊，因為矩形部分和播放器視窗區域交叉到，這些不屬於播放畫面的部分也會被以 JPEG 送去壓縮。

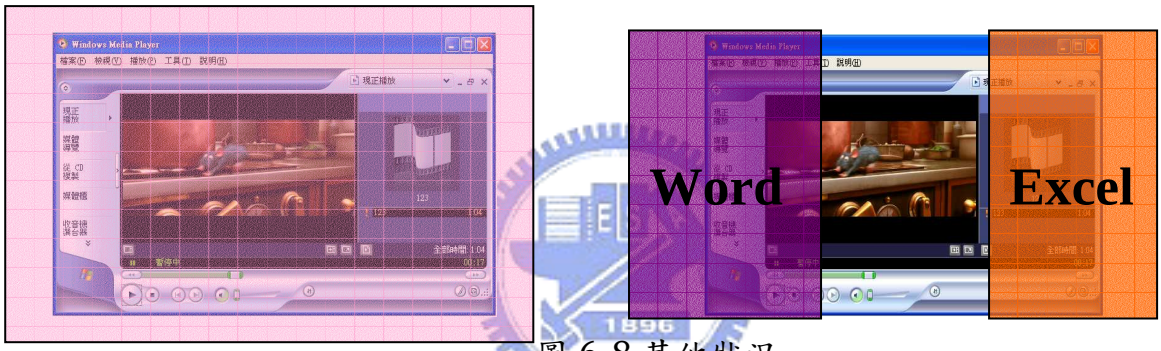


圖 6-8.其他狀況

●6.啟動 JPEG 或是以選定的 VNC 壓縮方式傳送畫面資料

下圖紅框中，說明怎麼樣判斷是否要啟用 JPEG 壓縮。一開始有兩個判斷式，第一個 jpegsign 表示 Client 是否支援有 JPEG，這部分在 Server 接收 SetEncodings 訊息中，判斷 padding 為 75 或 50 時，表示是由我們修改過的 Client 連線，此時會將此旗標設為 TRUE，表示 Client 支援 JPEG 壓縮。第二個 jpegrect 則是由上面一連串程序所啟用的，當播放器啟用且沒有被使用者最小化到工具列，組成 toBeSent 中的某個矩形也和播放器有交叉到，jpegrect 才會被設為 1，以上判斷都成立，就會呼叫 SendJpeg()函數來壓縮畫面資料並傳送。如果以上有一個沒成立，則 Server 會將矩形以原本選定的 VNC 編碼方式來編碼。如何掛 JPEG Library 請見附錄一。

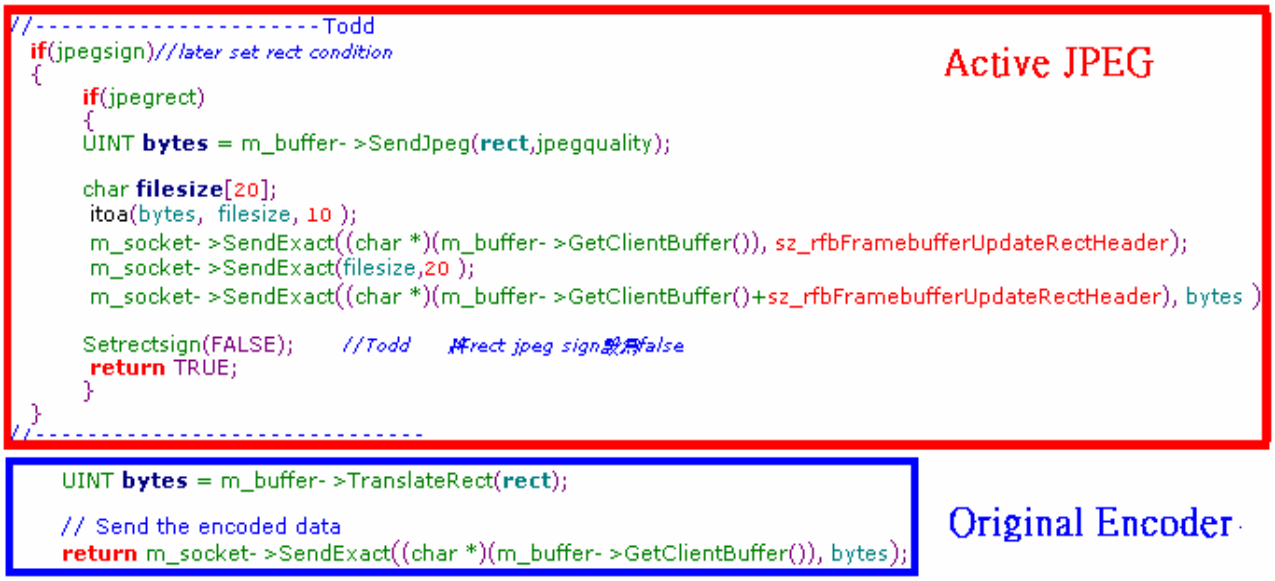


圖 6-9.啟動 JPEG 或原本 VNC 的壓縮

●改善後實際測試的流量統計

套用此節前面說明的，針對視訊部分進行 JPEG 壓縮的方法，進行實際的數據測試後，當在 JPEG 的 Quality 設為 75 的情形下，實際播放影片(長度 1 分 04 秒、672*272 24bits、23.976FPS)，此時它的總流量降為 8.9Mb，每秒的傳輸量只需 139kb！！和其它動輒上百 Mb 的總傳輸量，和每秒上 Mb 的流量比起來，大大改善了頻寬的部分。但這卻衍生了另一個嚴重的問題，雖然降低了網路傳輸量，卻讓視訊播放變得相當不流暢，令人難以接受，因此，是下半節要繼續深入探討的如何解決這問題。

表 6-3.改善後的流量

Encodings	每秒平均流量(Mb)	總流量(Mb)
Hextile	5.57	356.4
CoRRE	4.17	266.6
RRE	8.31	532
RAW	3.63	232.6
JPEG 75	0.139	8.9

6.2.2 衍生問題研究和解決

在上一小節中，說明了 Server 如何只針對視訊播放部分加入 JPEG 壓縮，但

這卻導致 Client 端播放畫面相當不順暢，這小節要進一步探討是何原因，造成這個棘手的問題。一開始先以實際測試時間來說明 Client 解完畫面資料封包、更新畫面到 Viewer 的時間，比 Server 從螢幕截取畫面、資料壓縮的時間要長許多，因此，問題很有可能是出在 Client 端身上；接著用 vncDesktopThread 流程圖輔助，詳細說明視訊播放不流暢的原因；最後發現是因為 Client 端解碼時用了 SetPixelV()函數，導致畫面更新速度太慢，造成視訊播放不流暢，為了改善此現象，提出了模擬 SetPixelV()函數的方法，並完全改善了此問題；最後以實際測試數據驗証。

●以數據說明 Client 端可能發生問題

下表中列出一部分畫面更新訊息中，Server 從桌面取出每個矩形畫面來壓縮花的時間，和 Client 接收對應封包並解壓縮，顯示到螢幕上花的時間來做比較，很明顯看出 Client 花費的時間比 Server 平均多了 3~5 倍，因此，合理懷疑 Client 某些部分出了些問題。在解決這個問題之前，先說明為何視訊播放畫面會不順暢。

表 6-4.壓縮和解壓縮時間

Number of rect	Compress Time	Decompress Time
1st rect	0.078	0.5
2nd rect	0.031	0.141
3rd rect	0.031	0.109
4th rect	0.031	0.109
5th rect	0.032	0.094
6th rect	0.031	0.14
7th rect	0.032	0.11
8th rect	0.031	0.109
9th rect	0.016	0.11
10th rect	0.032	0.125
11st rect	0.015	0.094
12nd rect	0.016	0.141
13rd rect	0.015	0.120
14th rect	0.016	0.115

●視訊播放不順暢的原因

播放畫面的不流暢造成的原因，主要是畫面更新的 Frame 數不夠所導致的。在第三章和第四章中都有解釋 VNC 是一套 Client Pull 的系統，而且，Client 是在接收並處理完一次畫面更新訊息後，才會再要求畫面更新，因此，當 Client 的處理速度比 Server 慢，要求畫面更新的次數降低，勢必會影響到 Server 傳送變動畫面資料量，Frame 數也因此而下降造成播放畫面的不順暢。

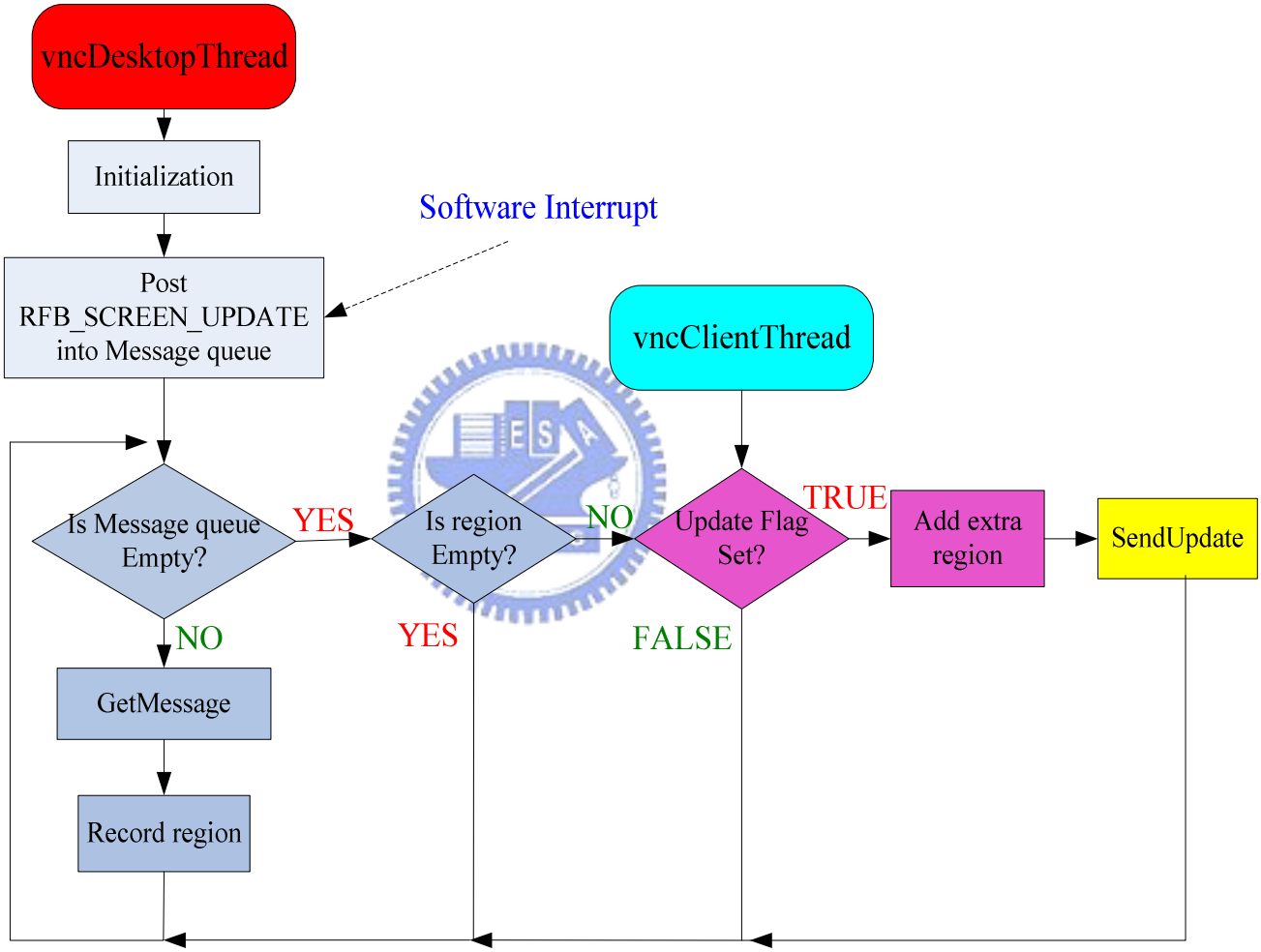


圖 6-10.vncDesktopThread

上圖是 vncDesktopThread 的流程圖(詳細運作過程請參閱 3.4.2 節)。假設 Server 目前正在播放影片，由 vncDeskThread 取得的初步範圍只有圖.所示，當 Update Flag 為 TRUE 時，才能再將額外的畫面範圍加入，但只有在 Client 要求畫面更新後，vncClientThread 才會將 Update Flag 設為 TURE，而 Client 送出畫面更新訊息只有在處理完上一次的畫面訊息後才會觸發，因此，一旦 Client 處

理時間加長，Update Flag 為 FALSE 的時間也會延長，進而影響 Server 傳送畫面的速率。假設 Client 處理一個畫面更新訊息時，Server 已經發生五次 Frame 變動，但因為 vncDesktopThread 前面部分(上圖斜線)，只會取得五次如圖.完全相同的範圍，等到 vncClientThread 送來畫面更新要求，Update Flag 被 vncClientThread 設為 TRUE，vncDesktopThread 才會額外將畫面範圍加入，進行處理，但因為目前 Server 螢幕的狀態已經是第 5 個 Frame，這也表示只會取到最後一個 Frame 畫面的資料，變相的將前 4 個 Frame 都丟棄了，因此，Client 也只能收到相當少的畫面資料量，這一連串的問題才造成播放品質如此差。

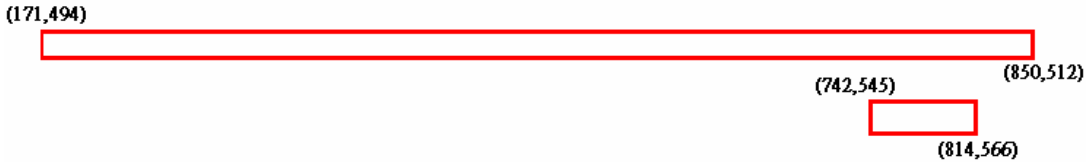


圖 6-11.初步範圍

●改善 Client 處理畫面更新訊息速度問題

由表.中數據，我們可以發現 Client 花費的時間比 Server 平均多 3~5 倍，也是造成視訊播放不流暢的主因，因此我們要細部進入解碼過程中找出是何原因。在分析了整個解碼程序後，我們發現有一個函數佔了整個解碼將近 80%的時間：SetPixelV()。

VNC Client 在解碼時會經常呼叫 SetPixelV()函數，它是 Windows 的 SDK 函數，用途是將解碼出來的畫面像素，設定在目的 Device Context 對應的位置上，雖然它可以直接和 Device Context 溝通，在程式撰寫上較簡潔直接，但效能卻非常差。因為呼叫 SetPixelV()函數時，要先做坐標軸轉換、剪裁區域判斷、將顏色匹配成每種不同 Device 能支援最接近的，最後還要根據不同的顏色格式進行運算，並將顏色寫入其所在位置，經過這麼多層內部運算處理才算完成，因此它的效率可想而知。

找出問題的癥結後，我們要針對這個函數來做改善，以另一種方法來模擬 SetPixelV()函數，將解碼後的畫面像素更新到 Client 的 Device Context 中。下圖中的紅框框即是用來模擬藍框框的函數，裡面做法是先將解碼後存在 m_zlibbuf 中像素資料，存成位元圖記憶體 m_HBitmap，再用 SelectObject()函數轉到相容的 Device Context-hSrcDC，接著用 BitBlt()函數複製畫面的資料到 Client 對應 Framebuffer 的 Device Context 裡。這一連串的程序都為了取代 SetPixelV()函數，雖然有些煩繁，不過改善後的成果相當令人驚豔，播放的品質有如即時呈現，而且測試後的網路流量也在可以接受的範圍中！[9]

```
//--------------Todd-------=SexPixels() function-----------------------------
UINT * pPixels = 0;
LONG lBmpSize = w * 1 * 4;
BITMAPINFO bmpInfo = { 0 };
HDC hSrcDC;
hSrcDC = CreateCompatibleDC(NULL);

bmpInfo.bmiHeader.biBitCount = 32;
bmpInfo.bmiHeader.biHeight = 1;
bmpInfo.bmiHeader.biWidth = w;
bmpInfo.bmiHeader.biPlanes = 1;
bmpInfo.bmiHeader.biSizeImage = w * 1 * 4;
bmpInfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);

HBITMAP m_HBitmap;
m_HBitmap = CreateDIBSection( hSrcDC, (BITMAPINFO *)&bmpInfo, DIB_RGB_COLORS, (void **)& pPixels , NULL, 0);
SetBitmapBits( m_HBitmap, lBmpSize,&m_zlibbuf[2048*4]);
SelectObject(hSrcDC, m_HBitmap);
BitBlt(m_hBitmapDC, x, y+dy, w, 1, hSrcDC, 0, 0, SRCCOPY);
DeleteDC(hSrcDC);
DeleteObject(m_HBitmap);
//------------------------------------------------------------

//SETPIXELS_NOCONV(&m_zlibbuf[2048*4], x, y + dy, w, 1);
```

圖 6-12.模擬 SetPixelV()函數

下表說明針對視訊播放區域做 JPEG 壓縮，和取代 SetPixelV()函數後，分別所統計出來的流量，明顯地比之前使用 SetPixelV()函數時的流量要大，這是因為以模擬 SetPixelV()函數的方法，使得 Client 處理更新訊息速度加快，也直接增加處理訊息後去觸發畫面更新要求的次數，Server 能傳送過來的畫面更新訊息也相對提高許多，因此，這些流量增加是屬於良性的，雖然比前面的多一些，但還在可以接受的範圍。

表 6-5.模擬 SetPixelV()後的流量統計

Encodings	每秒平均流量(Mb)	總流量(Mb)
JPEG 75(Previous)	0.139	8.9
JPEG 75(Improved)	0.516	33

6.3 最後數據分析比較

將 VNC 內部壓縮方法 Hextile、CoRRE、RRE、RAW、在 5.2 節中提出的針對視訊播放區域做 JPEG 壓縮，和改善 SetPixelV()函數後，以實際影片播放測試(長度 1 分 04 秒、672*272 24bits、23.976FPS)，得到的所有數據，整理在下表中。

表 6-6.所有方法的流量統計

Encodings	每秒平均流量(Mb)	總流量(Mb)
Hextile	5.57	356.4
CoRRE	4.17	266.6
RRE	8.31	532
RAW	3.63	232.6
JPEG 75(Previous)	0.139	8.9
JPEG 75(Improved)	0.516	33

上表是和 VNC 內建的壓縮方法比較，為了有所區別，我們再以其他市面上的遠端桌面程式來做比較。下表是用相同影片檔案測試，以 Windows 內建的遠端桌面程式 RDP 和我們改善的 VNC 做比較，很明顯地，因為 RDP 也是採用不失真的壓縮方式，因此它的流量也是相當高。

表 6-7. RDP 和 VNC 的流量統計比較

Encodings	每秒平均流量(Mb)	總流量(Mb)
RDP	3.44	220
JPEG 75(Improved)	0.516	33

由以上表格中數據顯示，VNC 4 種壓縮和 Windows 的 RDP，他們都是以不失真的方式來壓縮和傳送畫面資料，每秒的傳輸量皆以數 Mb 等級計的，這是目前一般家庭網路速度所不能負荷的量。很明顯地，我們提出的方法中，雖然是用破壞性的 JPEG 壓縮，取代原本 VNC 提供的不失真壓縮，但 JPEG 對於圖片壓縮並不會有太大影響，因此，在可以接受的情形下，我們額外加入一些邏輯判斷，盡量只針對視訊播放的範圍做 JPEG 壓縮，如此 Client 端既能獲得相當的播放和使用品質，又可大幅降低網路流量，對於一般家庭網路速率普遍不快情形下，它也能夠保持相當不錯的播放品質。

第七章 結論

7.1 結論

本論文以 VNC 為主軸，前五章詳細介紹完整遠端桌面程式的成形，在第六章則提出方法來改善視訊播放時網路流量會暴增的問題。

第三章介紹 Server 的整個架構，說明如何接收 Client 的連線要求和溝通訊息。在偵測變動畫面範圍部分花了最大篇幅，透過 Windows 的 API-Hook 先取得初步變動範圍，再用判斷機制額外加入可能變動部分，縮小可能需要的比對範圍，最後只針對這些可能範圍進行比對，取得真正的變動範圍並壓縮傳送。第四章則是換到 Client 端，詳細說明了它的運作流程和如何跟 Server 互動。

在瞭解了 Server 和 Client 各自的工作原理後，第六章中則是探討每個遠端桌面程式共有的問題：播放視訊檔案會造成網路流量暴增，先從第三章中變動畫面處理過程延伸，針對這問題進行剖析，說明前因後果，再提出解決方法：針對視訊播放部分進行 JPEG 壓縮，當變動部分落入播放器才用 JPEG 壓縮，沒落入則延用原本設定的編碼方法，最後以實際測試數據來和未改善前的狀況比較，以我們提出的方法實作後，確實將傳送網路流量降為約原本的十分之一。

7.2 未來發展

在第六章中，我們針對視訊播放區域加入 JPEG 壓縮後，對於網路流量的問題有大幅改善，不過仍有兩個可以改進的空間：

●更精準的定位視訊播放區域

我們是先取得播放器的視窗位置，只要需更新的矩形區塊位置和它有交叉

到，都會以 JPEG 壓縮，但這簡略的判斷方式會導致許多不該被壓縮的地方也被壓縮，因此，可以再設計一個更嚴謹的定位方式來進行判斷。

●以壓縮率更高的演算法替代 JPEG

目前有許多影片的壓縮方法，如 MPEG、MPEG2、MPEG4、AVI、H.264 等…，如果以其他壓縮方式替代 JPEG，應該可以更進一步降低傳輸的資料量。

參考文獻

[1] Tristan Richardson, “The RFB Protocol”, ORL/AT&T Labs Cambridge, June, 2008

[2] Http://www.rdesktop.org/

[3] Oliver Jones 著，Introduction to the X Window System，黃豐隆譯，松崗電腦圖書資料股份有限公司，民國 82 年，台北

[4] 李宗學，「終端服務的桌面影像壓縮」，國立交通大學,碩士論文，民國 96 年

[5] 林隆煥，視窗程式設計函式庫 WIN 32 API，金禾資訊股份有限公司，民國 93 年 11 月

[6] Http://msdn.microsoft.com/en-us/default.aspx

[7] Charles Petzxold, Programming Windows, Fifth Edition, Microsoft Press, November 11, 1998

[8]http://www.microsoft.com/china/community/program/originalarticles/techdoc/hook.mspx

[9] Http://www.cndw.com/tech/program/2006042152596.asp

[10] Http://www.jpegcameras.com/libjpeg/libjpeg.html

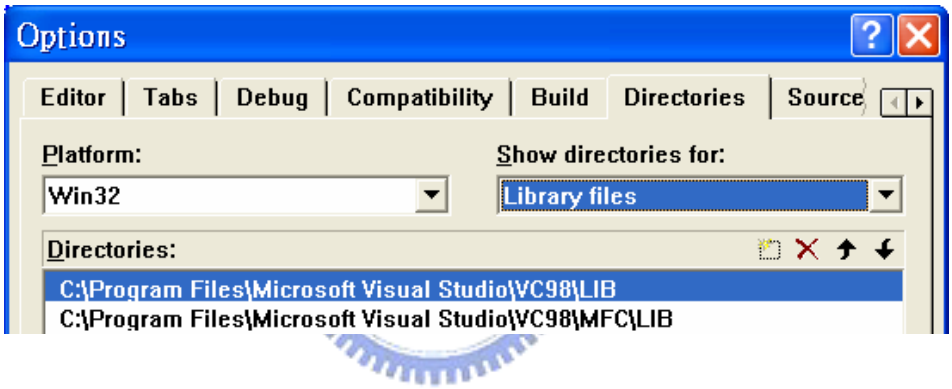
附錄一 ：如何掛上 IJG JPEG Library

●Setp1：取得 JPEG Lib

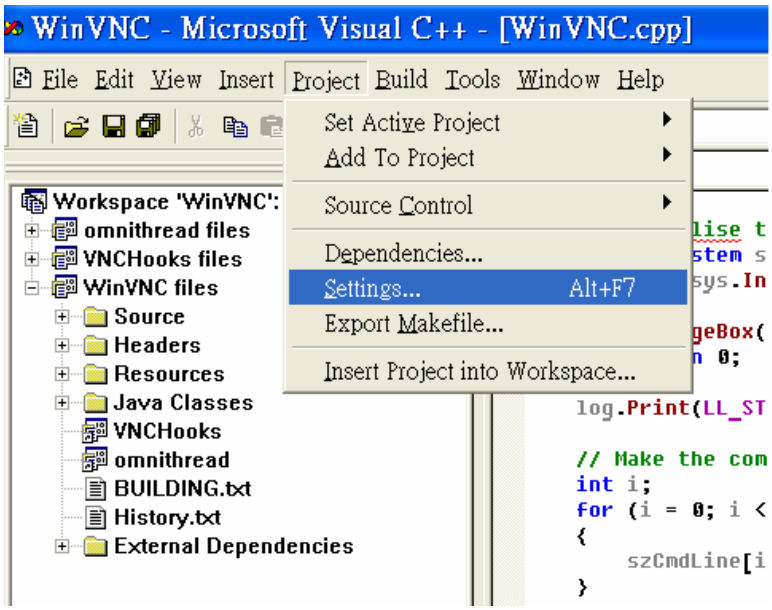
由 http://www.ijg.org/ 網站上，可以取得 IJP JPEG Library，只要將此 Library 整個 build 起來，即可取得"JpegLib.lib"。後續只要將它掛入自己的 Project 裡即可。http://www.jpegcameras.com/libjpeg/libjpeg.html 有詳細說明要如何使用這個 Library，在此就不贅述。

●Setp2：將 JpegLib.Lib 掛入 Project

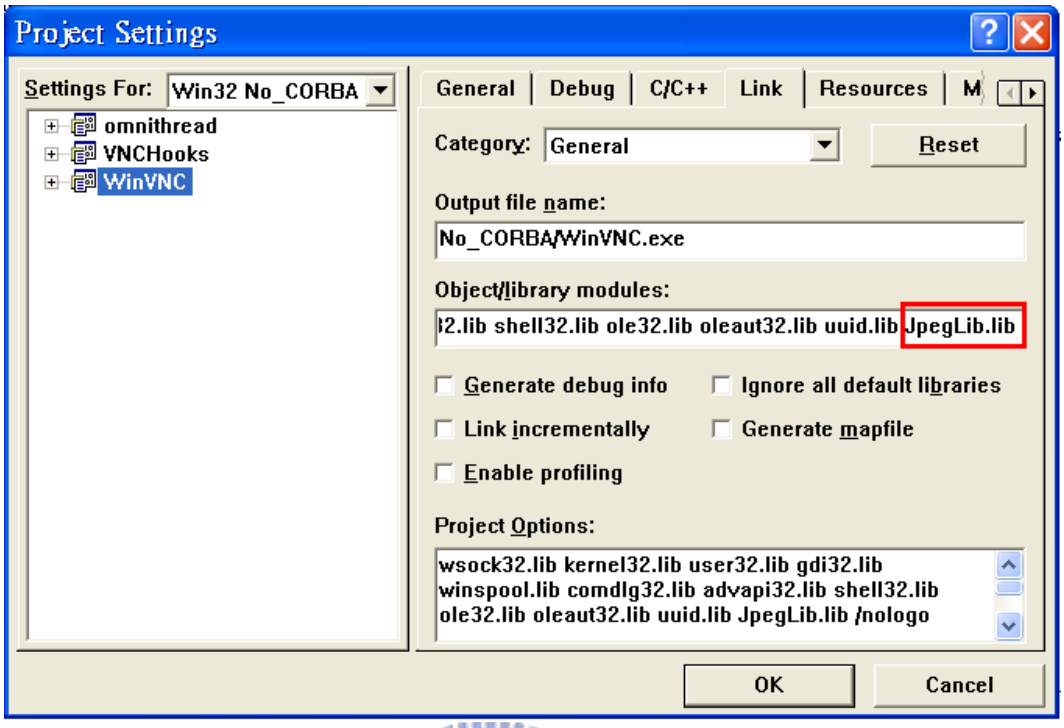
先將 JpegLib.Lib 複製到 Visual C++內定的 Library 目錄下。



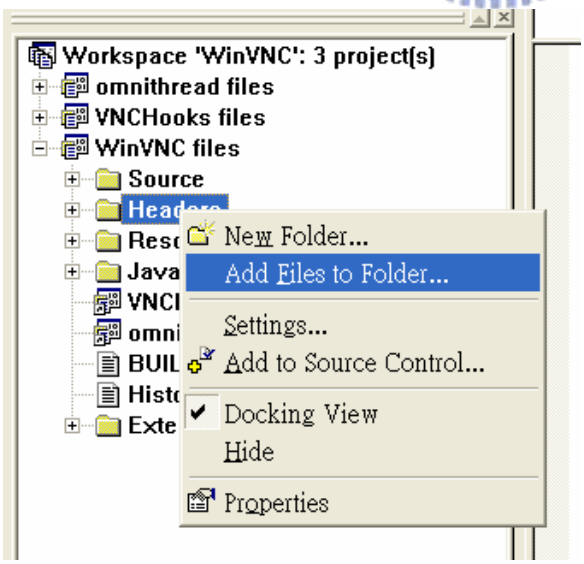
在 Project 的 Settings 中，include JpegLib.Lib。

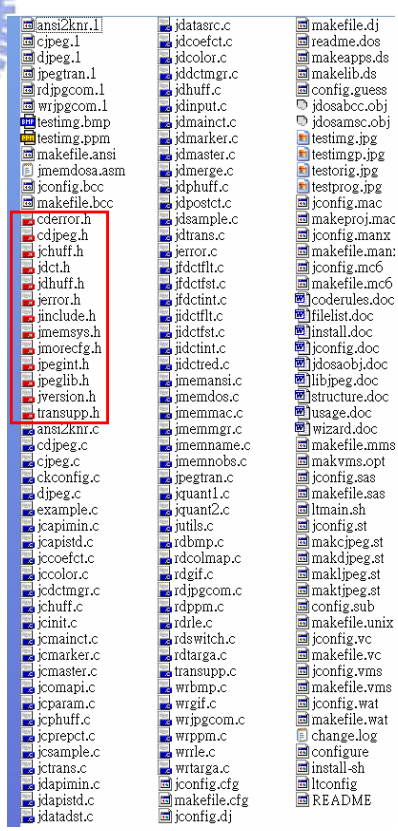


進入 Settings 後，在 Link 標籤的 Object/library modules 中填入 JpegLib.lib，



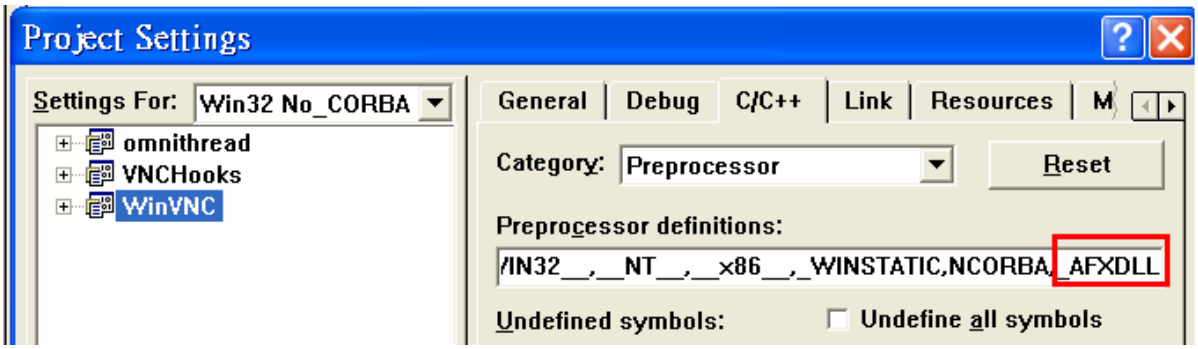
●Setp3：將 JPEG 中 Header 檔全部加入 Workspace





因為 Library 檔是由 c 檔所 build 起來的，因此，我們只要透過 Header 檔即可和裡面的函數做連結，因此，只需要將這些標頭檔加進我們的 Project 中。

Setp4：Preprocessor definitions 中加入,_AFXDLL



此選項是必須的，否則 Compile 時會出現 error。

Setp5：細部程式撰寫

可參閱 IJG JPEG Library 提供的 Example.c 檔，裡面有詳細的使用過程，在此就不再贅述[10]。