行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※
※　　　　　　　　　　　　　　　　　　　　　　　※
※　　網路醫學資訊系統之可擴充性與容錯機制　　　※
※　　　　　　　　　　　　　　　　　　　　　　　※
※　　　　　　　　　　　　　　　　　　　　　　　※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：□**個別型計畫**　　■**整合型計畫**

計畫編號：NSC　89－2219－E－002－029－

執行期間：89 **年** 08 **月** 01 **日至** 90 **年** 07 **月** 31 **日**

計畫主持人：郭斯彥

共同主持人：

本成果報告包括以下應繳交之附件：

□**赴國外出差或研習心得報告一份**

□**赴大陸地區出差或研習心得報告一份**

□**出席國際學術會議心得報告及發表之論文各一份**

□**國際合作研究計畫國外研究報告書一份**

執行單位：台灣大學電機工程學系

中　華　民　國 90 **年**　10 **月** 30　**日**

一、中文摘要：

關鍵詞：寬頻網路、醫療服務、分散式物件技術、可靠度、可擴充性、容錯機制

由於寬頻網路技術的日趨成熟，使得許多服務得以延伸到網際網路的環境。本計畫「網路醫學資訊系統」正是一個很好的實例，讓醫療服務得以藉由無遠弗屆的網路系統遍佈到世界各地。如此一來，病人可以在家裡面上網掛號候診，而醫生則透過網路對病人進行診斷，並可從遠端資料庫調閱所需的相關資料，如病歷或醫學影像等等，來協助診療的進行。換句話說，這就是「網路醫院」的具體實現。而在實作上，我們採用目前正在蓬勃發展的分散式物件技術來架構網路醫學資訊系統，除了能增加系統服務效率之外，也將有助於系統的擴充與維護。

目前有數個分散式物件的相關技術，而我們選擇使用 Microsoft **提出的** DCOM (Distributed Component Object Model)**來實作。雖然選用** DCOM**，我們亦研究如何善加利用各種技術的長處以期達到最大效益。在此子計畫執行過程中，我們已將這些分散式物件技術加以深入探討，從中獲得了各種技術的關鍵知識，同時配合著手實作初步的網路醫學資訊系統。我們除了獲得了以** DCOM **技術為基礎的程式設計經驗，也從此一系統的實作過程中進一步探討分散式物件在寬頻網路上的各項議題，如服務效率、系統可靠度、系統可擴充性、容錯機制等等。**

未來的網際網路應用將以分散式物件為基礎，工業界目前在這方面的進展非常快速，因此我們學術研究必須在這個熱門的領域投入人力與心力，迎頭趕上。藉由這個具有容錯能力及可擴充性的醫學資訊系統之發展，我們從中獲得以分散式物件技術為基礎之系統實作經驗，為將來其他寬頻網路系統服務之發展奠定基礎。

二、英文摘要：

Keywords：broadband network, medical treatment service, distributed object technology, reliability, scalability, fault tolerance mechanism.

Due to the steady progress on broadband network technology, it becomes feasible to extend many services to the networked environment. This project, a network medical information system, is one of the good examples, that makes medical treatment services ubiquitous through the widely-spread network systems in the world. In this way, patients can register to a hospital at home through the Internet, and doctors also give medical care through the Internet. At the same time, doctors can fetch related data such as case history or medical photographs placed in remote database systems on the Internet to help making diagnoses. In other word, this is the concrete realization of a “networked hospital”. We choose the distributed object technology to implement the system. It not only increases system efficiency but also is of great help in system extensibility and maintenance.

There are a few competing distributed object technologies and we choose: DCOM (Distributed Component Object Model) by Microsoft to implement the system. Although DCOM is used, we also try to make use of the advantages in other technology to maximize system performance. During the execution process of this sub-project, we studied these distributed object technologies in depth, and acquired much useful knowledge from them. After that, a simplified version of the network medical information has been implemented in the whole project. We not only acquired much experience of DCOM programming but also researched issues such as service efficienty, system reliability, extensibility, and fault tolerance mechanisms during the development of the system.

It is foreseen that Internet applications will be based on distributed objects in the near future. The progress on this topic is very fast in the industry, and we have to keep pace with it in the academia. With the development of the medical

information system on the Internet, we have acquired useful experiences of the implementation details using distributed object technology, and the project will build up a solid foundation on the development of other broadband network services.

三、研究背景及目的：

近幾年來，隨著網際網路以及有線、無線通訊技術的不斷發展與進步，網路使用者的數量與日俱增，架構在網際網路上的資訊服務項目也愈來愈多。為了讓廣大的社會民眾擁有更便利、更快速的醫療服務，「網路醫院」的概念於是誕生。有了網路醫院，醫療服務的提供不再只侷限於醫院本身。透過 Internet，病人可以隨時上網掛號候診，醫生也可立即對病人進行診斷，同時透過網路取得病人的病歷或 X 光片等相關資料以協助診療。除此之外，網路醫療資訊系統也具有教育的功能。醫生可以隨時隨地取得最新的醫療資訊，專業知識由此迅速傳遞，提供了一個便捷的進修管道。對於一般大眾，則也可從中獲得許多有用的醫學常識。

本計畫所欲建立的「新世代網際網路上醫學資訊系統」與現有的網路醫院比較起來，二者有許多特性是不一樣的。第一點，本計畫強調系統服務提供的即時性，病人可以透過網路直接與醫生進行互動，而身處各地的醫師們則直接在線上進行會診。第二點，本計畫的目的是希望整合全民的健康資訊，建立完整的資料庫系統，並且實作快速有效的系統可擴充機制，以提供高效能的整合性醫療服務。第三點，本計畫的實作將採用新興的分散式物件技術，能夠使得資訊系統的發展與維護更具有彈性，而這正是本子計畫的最終目標。

若將現今的電腦產業的發展分成軟、硬體兩方面來看得話，我們會發現硬體的發展與進步速度十分的驚人。雖然軟體業也經常有許多創新的發明或發展，但是其更新週期遠不如硬體來的快。於是乎，軟體業者與學術研究機構開始尋求這個問題的解答。他們發現硬體之所以進步如此快的其中一項重要因素就是硬體在設計製造的過程中，硬體設計師運用了大量的模組化設計技巧以避免增加新功能時複雜度的增加。例如，在設計過程中若使用模組化的設計技巧，則在發現錯誤的時候可以很快的找出錯誤的所在位置並加以除

錯更正（因為各模組之間相互獨立）。可是若不使用模組化的設計技巧，則在發現問題的時候即使已經找的問題的所在位置，修改該處並不能保證整個設計就完全沒有錯誤。這主要是由於每一個問題都糾纏在一起，某個地方出現錯誤可能是許多地方所造成的結果（當然也可能只有發生問題的位置所造成）。此外，若使用模組化的設計方法，則問題將會被切割成較小的子問題。這有助於我們對這些子問題的掌握，進而更容易解決整個大問題。另外一個由使用模組化的設計技巧所獲得的好處是各個模組的重複使用。由於每個模組之間完全相互獨立，所以我們可以拿一個模組來和另外一個模組組合使用也不必擔心會有問題。由於這些優點，使用模組化的設計技巧將能夠增加生產率。

在軟體方面來說，模組化的設計就是使用物件導向的概念。每個物件都是都可以重複使用，並做到互相獨立。但是在一個分散式的環境中，單純只有物件導向概念所撰寫出來的程式碼仍然不能滿足我們的需要。例如，我們希望客戶端的程式在改寫之後並不需要重新編譯伺服端的程式。我們也希望不管客戶端或伺服端如何的更改都不會互相影響，而且不會受到其實作的變更而必須重新改寫。也就是說這需要物件能夠動態的被載入執行。要物件能夠被動態地載入記憶體中執行，物件就必須是一個以二位元可執行碼存在的實體。物件也必須要能夠動態連結到使用物件的程式碼中。當然，在分散式的運算架構之下，我們還需要物件能夠有網路透通性。所謂網路透通性就是，客戶端可以不管元件的位置在哪裡；只要客戶端想要執行，那麼它就可以取得元件來使用而不需要特別針對元件的位置來撰寫程式碼。這些要求就是撰寫一個軟體元件的準則。簡單來說，軟體元件模型是達成（可抽換）軟體 IC 的撰寫程式準則，而不是一種新的程式語言。

目前在分散式物件技術相關領域中有三個主要的競爭者，它們分別是由 Microsoft 提出的 DCOM (Distributed Component Object Model) 技術，Object

Management Group 提出的 CORBA (Common Object Request Broker Architecture) 技術，以及由 Sun Microsystems 主推的 JAVA RMI 技術。經過評估，我們以微軟的 DCOM 為基礎來架構這個網路醫學資訊系統，再分別採用其他分散式物件技術的優點作為輔助，探討分散式物件在寬頻網路上的各項議題，如服務效率、系統可靠度、容錯機制等等。我們仔細研讀了 COM/DCOM、CORBA 等分散式物件技術的規格以及其運作方式，從中獲得了各項技術的關鍵知識。同時，我們依據 DCOM 技術的規格實際撰寫了一些應用程式，並觀察程式的執行流程，以深入瞭解 DCOM 的運作機制。此外，我們也研究了分散式系統的可擴充性、系統效能、與容錯機制等議題。配合其他子計劃的需要，著手實作具有初步功能的網路醫學資訊系統。這些研究有助於讓系統效能提昇、可靠度增加、以及未來的擴充性提昇。我們除了獲得了以 DCOM 技術為基礎的程式設計經驗，也對分散式的運算環境有了更深入的了解，預期對將來繼續擴充系統具有相當之幫助。

分散式計算已經成為主流，這是因為在高速網路的進步及網際網路的蓬勃發展下，加上物件導向程式也已成為發展可重用性軟體的主流，分散式物件技術結合這兩大趨勢而漸漸受歡迎。愈來愈多的系統軟體建立成分散式物件的應用程式讓彼此間能分享許多共同的目標，這個計畫主要的目的是探討 COM/DCOM 的特性並將分散式物件技術加以應用在網路醫學資訊系統中，以達成高效率、可擴充性及容錯機制的實現。同時我們也會配合其他子計畫的需要，從旁協助採用分散式物件技術來發展軟體元件，如此一來將可使得系統未來的維護更容易、更快速。

四、研究方法及成果：

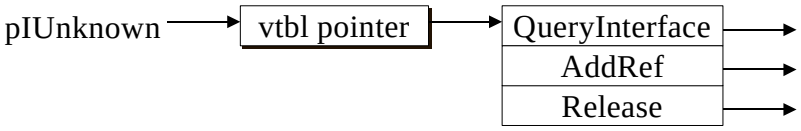
我們在前一節提到軟體的模組化，每個物件都是可以重複使用，並且彼此互相獨立。要達到此一目標，物件必須是一個以二位元可執行碼存在的實體。另外，還需要物件具有網路透通性以滿足分散式物件的需求。當我們要求軟體要能夠滿足上述需求的時候，當然就必須要有一些限制才可以達到我們的要求—將物件獨立出來。這些也就是軟體元件模型中所要求的“界面”與“動態連結”兩個要素所要達成的重要目標。在下文中，我們將有部份篇幅用來討論如何將用戶端程式與元件的關係切割開來。然後再討論如何提升生產力，也就是程式碼的重複使用。以下繼續針對分散式物件技術的原理詳細說明，大部分都以 COM/DCOM 為例。

在軟體元件模型中，做每一件事情都是由界面出發。在載入元件之後，若我們要使用某一個界面，則必須經由界面的詢問才可以獲得所要的界面，進而使用該界面所提供的服務。DCOM 軟體元件模型要求所有的元件都必須要繼承 IUnknown 界面。經由使用 IUnknown 界面中的 QueryInterface()函式可以用來詢問並獲得其它界面。在獲得了所需的界面之後，就可以使用該界面所提供的任何服務。以下為 IUnknown 界面的宣告：

```
#define interface struct
interface IUnknown {
    virtual HRESULT __stdcall QueryInterface(const IID&iid, void** ppv) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
};
```

值得注意的是 IUnknown 界面為純抽象類別(Pure Abstract Class)。純抽象類別就是只有純虛擬成員函數(Pure Virtual Member Function)而沒有任何的資料成員(Data Member)。此外，IUnknown 界面中的三個函式的宣告順序非常的重要，因為它影響了這個界面在記憶體中的結構。以上面的 IUnknown 界

面的宣告為例，它的記憶體結構如圖一所示：

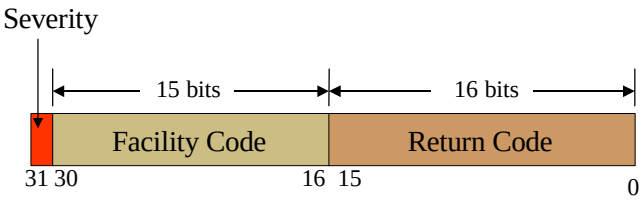


圖一： IUnknown 的記憶體結構

由於元件必須繼承 IUnknown 界面，所以元件必須要實作自己的 QueryInterace()來讓用戶端程式使用它所提供的服務。在繼承 IUnknown 界面的時候必須要使用公用(public)繼承的關係而不能夠使用虛擬(virtual)繼承，否則整個記憶體結構將會與軟體元件模型所要求的不同。實作 QueryInterface() 的時候，要特別注意第一個參數指定我們所要的界面的識別字串(GUID)。這個 GUID 必須是獨一無二的，這樣才不會造成混淆而不知道所要求的是哪個界面。由於所有的動作都是從界面出發，故我們必須對獲得界面的 QueryInterface()函式有所限制才能夠達到程式順利運作的目的。以下列出對 QueryInterface()的要求：

1. 用戶端程式每一次取得的都是相同的那一個 IUnknown 介面。
2. 用戶端程式可以取得任何曾經取得過的介面。
3. 用戶端程式可以取得已經擁有的介面。
4. 用戶端程式可以由某個介面回到起始介面。
5. 如果用戶端程式可以從某個介面取得一特定的介面，那麼就可以從任何介面取得該介面。

QueryInterface()函式的傳回值型別為 HRESULT (如圖二所示)。由於代表成功與失敗的代碼分別有許多個，我們必須要使用 SUCCEEDED()與 FAILED()兩個巨集來判斷詢問介面是否成功。

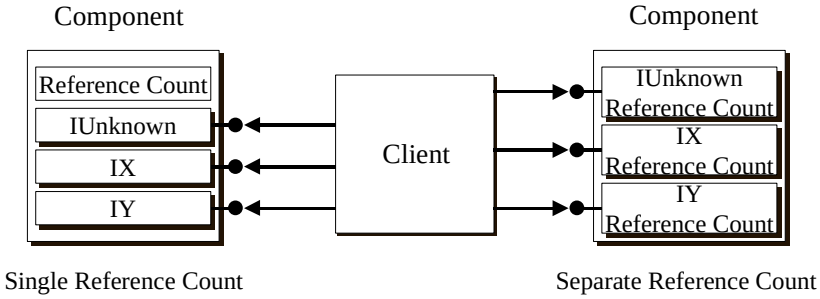


圖二： HRESULT **格式**

前面有提到 GUID **這個界面（或元件）的識別字串。這個** GUID **是從** OSF (Open Software Foundation)**的** DCE (Distributed Computing Environment)**中借來的東西。它不管是在空間上或時間上都必須是要獨一無二的。也因為這個緣故，所以** GUID **是一個非常大的資料結構—它佔有** 128 **個位元。故為了效率的考量，我們在需要** GUID **的地方都是以傳位址的方式來達成。**

將程式切成元件除了上述的好處之外，還有另外一個好處就是節省記憶體與磁碟空間。若有數個用戶端程式需要使用相同的元件，那麼該元件只需要一份存在於記憶體或磁碟中，而所有的用戶端程式可以共享元件。當用戶端程式不再需要元件的時候，如果能夠將元件從記憶體中釋放掉的話就更能夠善用記憶體空間。因此，我們需要一個用來管理元件生命週期的方法。當我們在使用一個元件的時候，我們可能使用了這個元件所提供的數個介面。我們不希望某個介面不再需要但其它介面仍在使用的時候釋放掉該元件。若要由用戶端程式來控制元件的生命週期的話，則用戶端程式必需要知道那些介面是屬於同一個元件、何時可以釋放掉元件。這種做法在介面與元件數目不多的時候還算勉強可行。但是隨著介面或元件的數目增加，這個方法就越來越不可行。對於這個問題，最好的解決之道就是由元件自己管理自己而不是由用戶端程式來管理。因此，每個界面都要有一個參用計數(Reference Count)**，用以記載目前有多少個客戶程式正在使用該界面。當我們欲使用某一個界面所提供的服務的時候，我們就必須要遞增參用計數，以便告訴元件我們正在使用某一個界面。這就是** IUnknown **界面中的** AddRef()**所做的事。當我們不再需要某個界面的服務的時候，我們可以藉由呼叫** IUnknown **界面中的** Release()**來遞減參用計數，以便告訴元件我們已經不再需要某界面的服務了。一旦參用計數遞減到零的時候，則元件將會自動將自己從記憶體中釋放掉。**

使用參用計數的時候要特別注意一件事，那就是使用哪個界面遞增參用計數就必須使用該界面來遞減參用計數。主要的原因是因為，我們並不知道撰寫元件的人是把元件實作成所有的界面都共享一個參用計數或者是每個界面都有自己獨立的參用計數。我們使用元件的人一定要將元件視為以每個界面都有自己的參用計數來實作，而不管真正的實作方式為何（如圖三所示）。因此，叫用哪個界面來遞增參用計數就必須使用該界面來遞減。這樣才不會釋放掉不該釋放的元件或界面，也不會有界面或元件沒有被釋放掉而一直佔據在記憶體中。最後一點關於參用計數的注意事項，用戶端程式絕對不可以依賴 AddRef()與 Release() 這兩個函示的傳回值。



圖三： 使用元件的時候，必須將元件視為每個界面有自己的參用計數

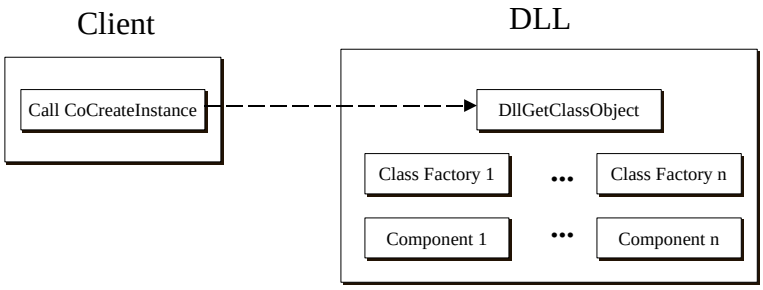
有了上面所提到的這些機制之後，還有一個重要的要素才能夠達到將元件與用戶端程式完全切割開來。這個不可或缺的要素就是動態連結。經由動態連結，用戶端的程式碼可以與元件完全獨立。不需要因為對方修改而重新編譯。此外，這個機制也是在達成網路透通性時用來連結用戶端與元件程式的要素。動態連結是用來將伺服端的程式與客戶端的程式連結。伺服端可分為行程內伺服端(in-process server)與行程外伺服端(out-of-process server)。DLL 檔又稱為行程內伺服端，EXE 檔又稱為行程外伺服端。以行程內伺服端為例，因為元件現在是在 DLL 中，所以第一要務就是從 DLL 中輸出產生元件的函式。要輸出函式有兩個步驟：

1. 為要輸出的函式加上 extern “c” 的宣告，以確保函式使用 C 的連結方式（避

免 Name Mangling）。

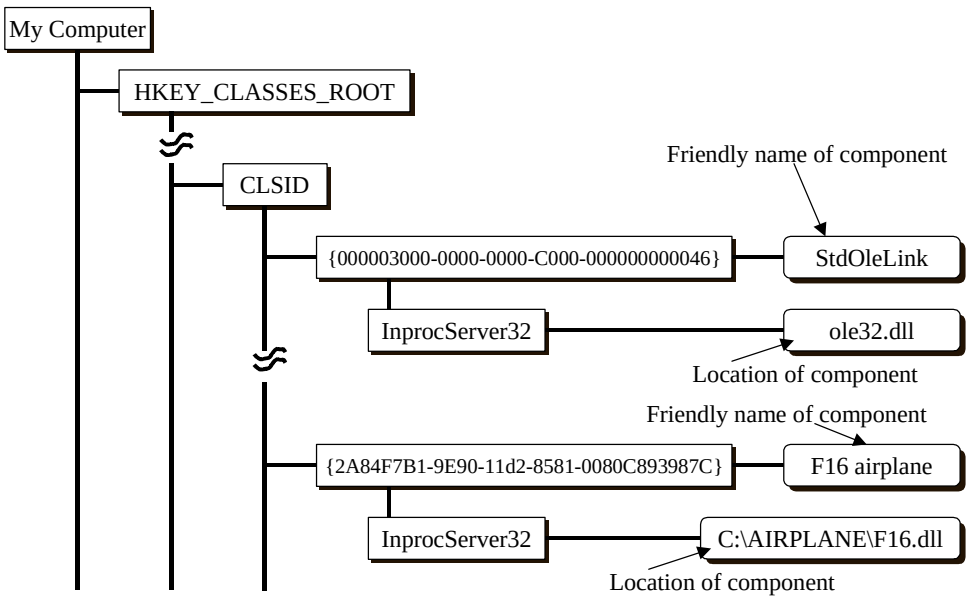
2. 使用 DEF 檔告訴連結器(linker)所要輸出的函式名稱。

欲使用行程內伺服端的服務時，先用 LoadLibrary()將 DLL 載入記憶體中，然後使用 GetProcAddress()來取得所要的函式位址。一個 DLL 檔中可以包含有許多的元件，所以元件並不等於 DLL（如下圖四所示）。



圖四：一個 DLL 可以包含許多個元件

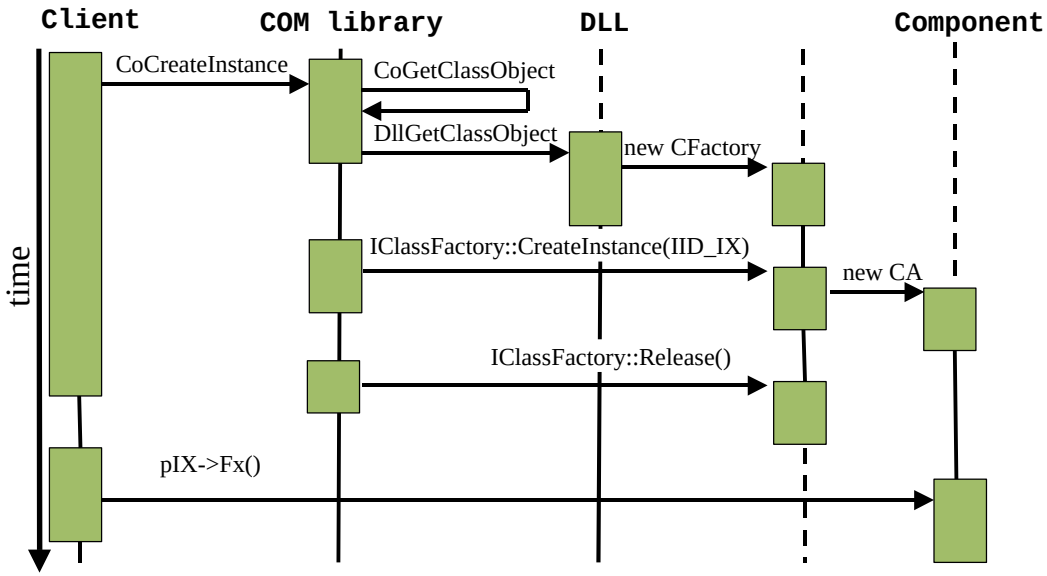
前面曾經提到過每個元件可以利用識別字串 GUID (或 CLSID)來加以辨識。而 GUID 這些訊息都存在“登錄資料庫(Registry)”中。Registry 的結構如圖五所示。我們可以使用 DllRegisterServer()與 DllUnregisterServer()分別來將 CLSID 的訊息放入 Registry 與移除自 Registry。



圖五：Registry 結構概觀

當我們使用元件的時候難免會遇上需要使用軟體元件模型函式庫(COM Library)中的函式。這時候我們就需要對軟體元件模型函式庫做初使化的動作，這可以經由 CoInitialize()函式來達成。不再使用的時候可以用 CoUnitialize()來結束。OLE 是架在元件模型函式庫之上，另外又加入剪貼簿、拖曳操作(drag & drop)、ActiveX 文件、ActiveX 控制元件、Automation 等等的支援。若欲使用這些功能的話，我們必須使用 OleInitialize()與 OleUnitialize()來做初使化與結束的動作。

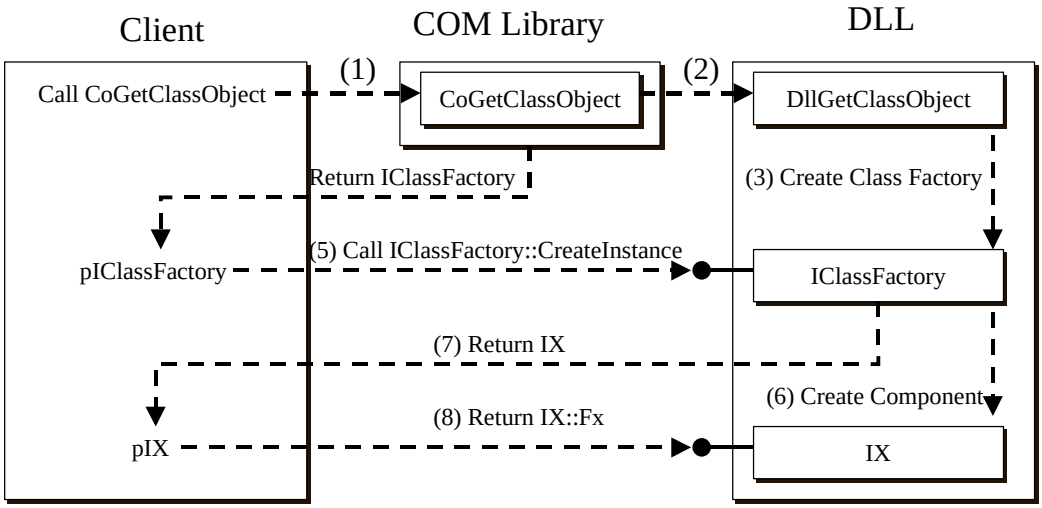
到目前為止，我們已經可以將用戶端程式與元件完全分開。不過，我們沒有多提到元件是如何產生的。其實元件的產生可以使用元件模型函式庫中的 CoCreateInstance()來達成。CoCreateInstance()的運作方式如圖六所示。



圖六：CoCreateInstance() 的運作方式

雖然 CoCreateInstance()這個函式可以滿足產生元件的最基本要求，不過這個函式有一些缺點。首先，若我們要一次產生多個元件的時候。我們需要呼叫此函式很多次，每次產生一個函示。每呼叫一次函式都會需要作一些額外的動作，這些動作會降低程式的效率。另外，我們或許需要對元件的產生過程加以控制，故撰寫一些自己的程式碼來做這件事情。這時候

CoCreateInstance()**可能就無法滿足我們的需求了。我們需要一個更有彈性的產生元件的方法，也就是改用類別工廠**(Class Factory)**。類別工廠其實也是一種元件，只不過它的作用比較特別。它是用來產生元件的元件。由於任何元件都必須繼承** IUnknown **界面，因此類別工廠也不例外。類別工廠是在元件的** DLL **中實作並輸出** DllGetClassObject()**以取得指向類別工廠界面的指標。而用戶端程式則是經由叫用** CoGetClassObject()**這個軟體元件模型函式庫中的函式來使用它。整個運作過程就如圖七所示。**



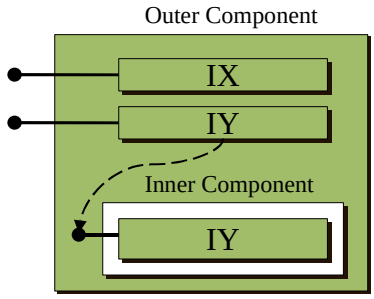
圖七：使用類別工廠來產生元件的詳細過程

在前面的圖五中，我們已經看到一個 DLL **可以包含許多的元件。由於類別工廠也是元件的一種，故一個** DLL **當然可以包含有許多的類別工廠。特別需要注意的是，某一個類別工廠只能產生特定** CLSID **所代表的元件。這就是圖五中一個類別工廠對應到一個它可以產生的元件的原因。**

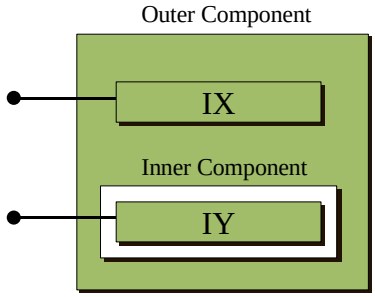
上面這些就是我們要將用戶端程式與元件分割所需要的步驟或方法。只要遵照這些原則就可以撰寫出符合軟體元件模型的可抽換的程式。在有了可抽換的能力之後，若發現有問題的元件我們就可以單獨的將其分隔出來。然後換以功能正確的元件。我們也可以針對某一個元件的功能加以增強，而不需要考慮其它元件（前提當然是各元件切割的非常的好且要相互獨立）。這

樣的話，在提升軟體功能的時候就不會浪費太多時間在找尋因為各部份糾纏而導致的（沒有必要的）錯誤。這些所解決的是提升軟體功能或找尋錯誤的時候效率不彰的問題。但是，當我們需要重新撰寫一個新軟體的時候，我們希望的是使用以前努力撰寫的程式碼。軟體元件模型並沒有提供一個像 C++ 一樣的實作面的繼承。軟體元件模型所提供的只是界面的繼承而已。所幸，軟體元件模型提供了兩個方法可以使用以前所撰寫過的程式碼。這兩個方法就是包含(Containment)和群集(Aggregation)。雖然使用這兩個方法所需要的勞力比使用簡單的 C++ 繼承來的多，但是換取到是具有前面所述功能的元件（動態連結、網路透通性…等）。所以這一些努力應該還是值得的。關於這兩個方法我們以一個例子來說明。假設今天您是一間電腦零售店的老闆，甲與乙是您的兩位員工。其中，甲已經在您的零售店中上班很久了，而乙卻是剛到您的店中工作。所以當有客戶上門要談生意的時候，您可以很放心的將工作交給甲。讓甲與客戶直接對談，因為他已經很熟悉所有的工作內容了。甲與客戶之間所談的交易您完全不需要去操心，甲可以將工作獨立完成。但是，反觀乙，由於他剛到您的店中上班沒有多久，所以對許多細節都不甚清楚。這時候您就必須要起個頭做一點，然後再把工作交給乙去做。等到乙做完了以後，您可能還需要再檢查一下。在這種狀況下的甲就像是軟體元件模型中的群集，而乙就像是包含。

以圖形來表示包含（見圖八）與群集（見圖九）的話，可以由用戶端是直接和內部元件接觸還是透過外部元件與內部元件接觸兩種情形來分辨這兩種方法。使用包含的方法時，我們只需要為需要包裝的界面撰寫包裝程式即可。但是這種做法雖然簡單，卻需要重新實作界面與額外叫用內部元件的麻煩。若使用群集的方法的時候雖然不必自己額外去呼叫內部元件，不過需要更動到比較多的部份。

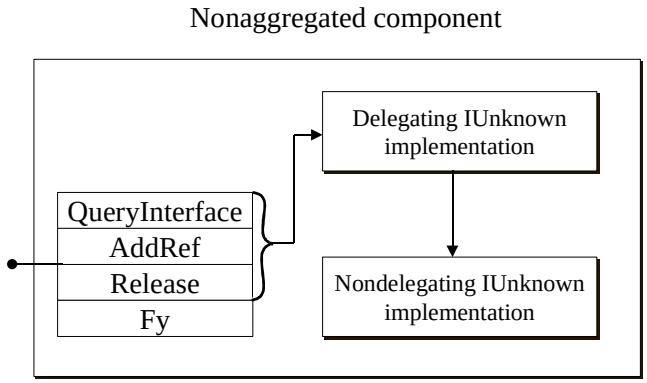


圖八：包含 (Containment)

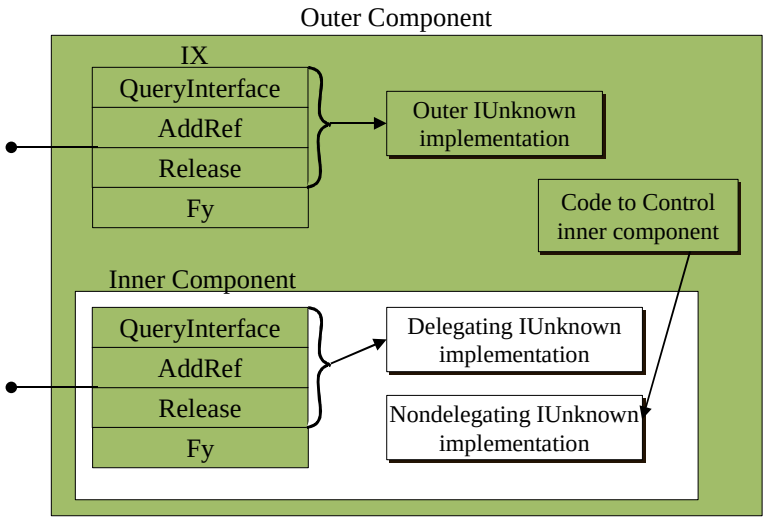


圖九：群集 (Aggregation)

為了支援群集，內部元件必須要實作兩套 IUnknown **界面。我們將他們稱為** delegating unknown **與** nondelegating unknown**。依據是否為群集狀態，** Delegating unknown **負責將用戶端程式對於** IUnknown **的要求委託給** outer unknown **或** delegating unknown **來處理。若內部元件沒有群集在其他元件中（此時應該不可稱之為內部元件才對，不過為了方便說明姑且用之），那麼** delegating unknown **就將用戶端程式的要求轉給** nondelegating unknown**，否則就轉交給** outer unknown**。對於使用群集元件的用戶端程式而言，它永遠是透過** delegating unknown **來使用整個元件, 只有外部元件才會透過** nondelegating unknown **來使用內部元件。圖十表示的是沒有群集的情況。圖十一則是在群集情況下，**delegating unknown **如何將用戶端的要求轉給** outer unknown**。**



圖十：非群集時，將要求轉給 nondelegating unknown



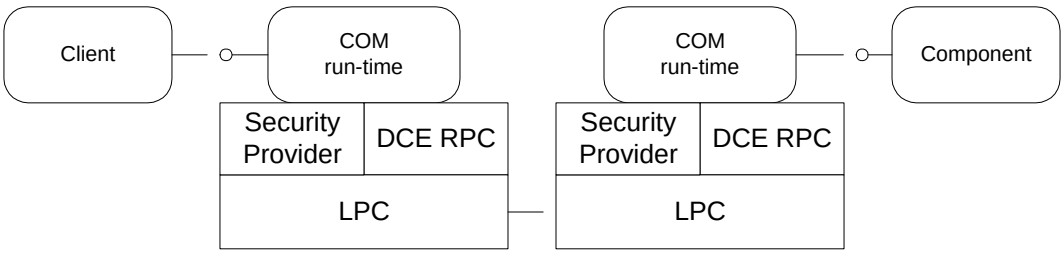
圖十一：群集時，將要求轉給 outer unknown

以上我們說明了 COM **的運作機制，現在我們將** COM **延伸到分散式的環境，也就是所謂的** Distributed COM (DCOM)**。**COM **定義了元件及其客戶端之間的互動。這個定義使得客戶端與元件之間的連接不需要任何居間的系統元件;客戶端可無額外負擔地呼叫元件所定義的方法**(method)**。下圖是** COM **元件在相同程序**(process)**的示意圖。**

圖十二：在相同程序的 COM **元件**

在現今的作業系統，不同的程序之間通常會彼此隔開。所以，若某客戶端需要與在另一個程序中的物件進行溝通，則需要使用作業系統所提供的某

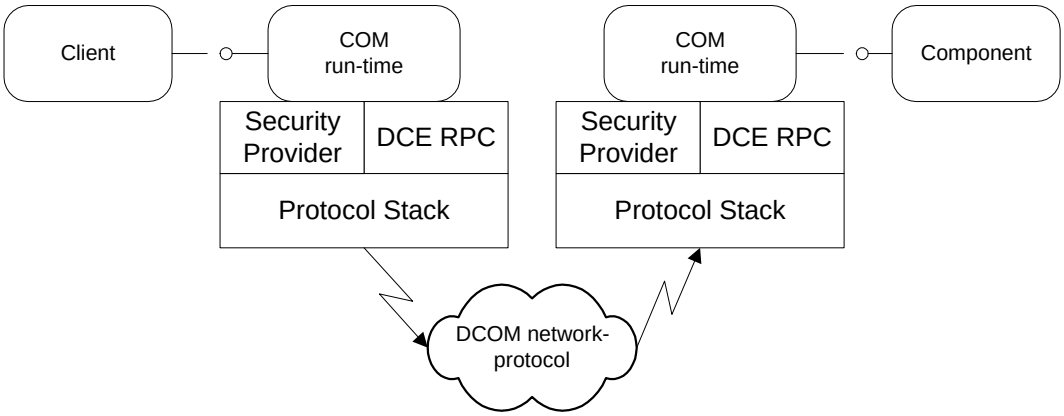
種程序間(inter-process)溝通機制。COM 提供了一套這樣的溝通機制，使得客戶端完全不須知道其中是如何運作的：它把從客戶端往元件的呼叫攔截下來，再將這個呼叫繼續傳遞給在另一個程序的元件。下頁的圖十三為上述說明的示意圖。



圖十三：不同程序中的 COM 元件。

當客戶端與元件不在同一台機器上的時候，DCOM 就以網路協定來把程序間的溝通機制給取代。客戶端與被呼叫的元件都將不會發覺它們兩者之間的距離變得遠了些。

圖十四是 DCOM 的整體架構：COM run-time 對客戶及元件提供了物件導向的服務；而使用 RPC 與 security provider 來產生適合於 DCOM 標準通訊協定的標準網路封包。



圖十四：DCOM 在不同機器上的 COM 元件

提供一個全方位的網路醫學資訊系統是項非常具挑戰性的工作，因為它

必須是不會停下來的服務，否則將會對醫生或病人造成重大影響，甚至危害生命安全。另外效能的表現對於此系統的成功也是非常重要的，要是不能滿足醫生或病人的期待，他們便會尋求其它方法。再者可擴充性也是必須包括的，因為系統必須能夠隨著醫院及病人之增加，處理大量同時連線的客戶請求。

原有的網路醫學資訊系統最缺乏的是在客戶端的發展，若是沒有改善並加強使用者介面，則與系統間的交談無法得到改進。而改進會受限於原來已存在的系統架構，若不加思索的修改必需完全改寫原有的程式，這就限制網路醫學資訊系統的發展。

所以，我們將研究的主要目標設定在如何提供簡單且有效的存取介面，如何將目前最新的多層式分散式技術應用於醫學資訊系統，並提供方法平穩轉移以主機為中心的程式到分散式計算的程式。進而提供可擴充性及容錯機制，使得客戶與伺服器都能感到效能提升及可靠度提高，使系統服務更無遠弗屆。

分散式元件物件模型的特徵包括了介面與實作的徹底分離，支援物件具有多重介面、語言中立、即時二位元可執行碼的重複使用、元件位置透通性、可延伸架構、間接存取、版本管理及伺服器生命週期管理。也就是說以 COM/DCOM 為發展分散式程式的平台，則研究者及研發者可以專心於對他們程式比較重要的課題, 而不需將大部分的努力投資在建立支援的基礎建設.

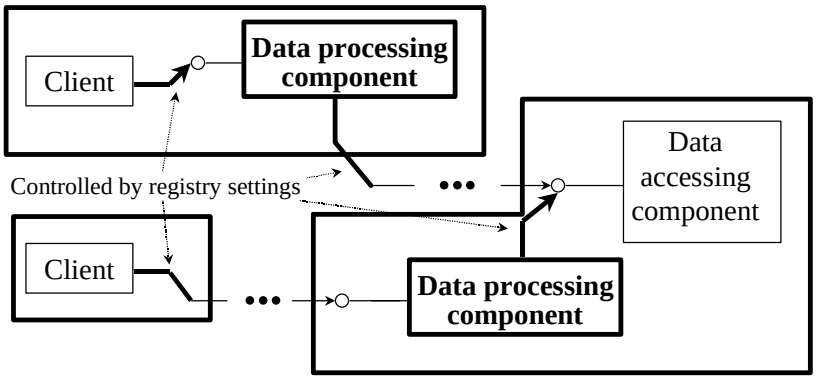
接下來，我們將具體說明 COM/DCOM 可以成為 R&D 建立分散式元件程式的平台的理由。以下從 COM 的主要特徵：透通性、延伸性、間接存取、更新版本及伺服生命週期的管理作說明。

(1) 透通性

在啟動時，COM 同時支授非透通及透通模式。在非透通模式中，客戶端

可以明確地指出伺服元件是否在 DLL 內或 EXE 內。甚至假如元件在遠端也可指出遠端機器名稱。相反地若選擇透通模式，則讓 COM 自行詢問登錄資料庫來決定以上屬性。一旦機器名稱確定，COM 將會利用已存在的物件實體來完成任務。若是物件實體不存在，則由 COM 自動找出伺服器實作檔產生一個新的物件實體。為了要在方法呼叫上提供透通性，伺服器僅傳回物件實體的參照(reference)。而物件參照內包裝了所有由客戶端到伺服物件之間所有的連線訊息。一般來說它包含了 IP 位址、埠號，以及物件位址。雖然伺服器通常會傳回在本身機器上的物件參照，有時也可以傳回由別台機器所得的物件參照。當物件參照傳到客戶端時，客戶端的 COM 函式庫便會解開它。根據此連線資料，傳回介面的指標，當客戶端透過這指標呼叫時，這個呼叫只傳給所指的物件介面而不會打擾伺服器其他物件。其實這也就是間接存取的應用之一。圖十五表示了元件位置的透通性。

- **Flexible deployment (location transparency)**

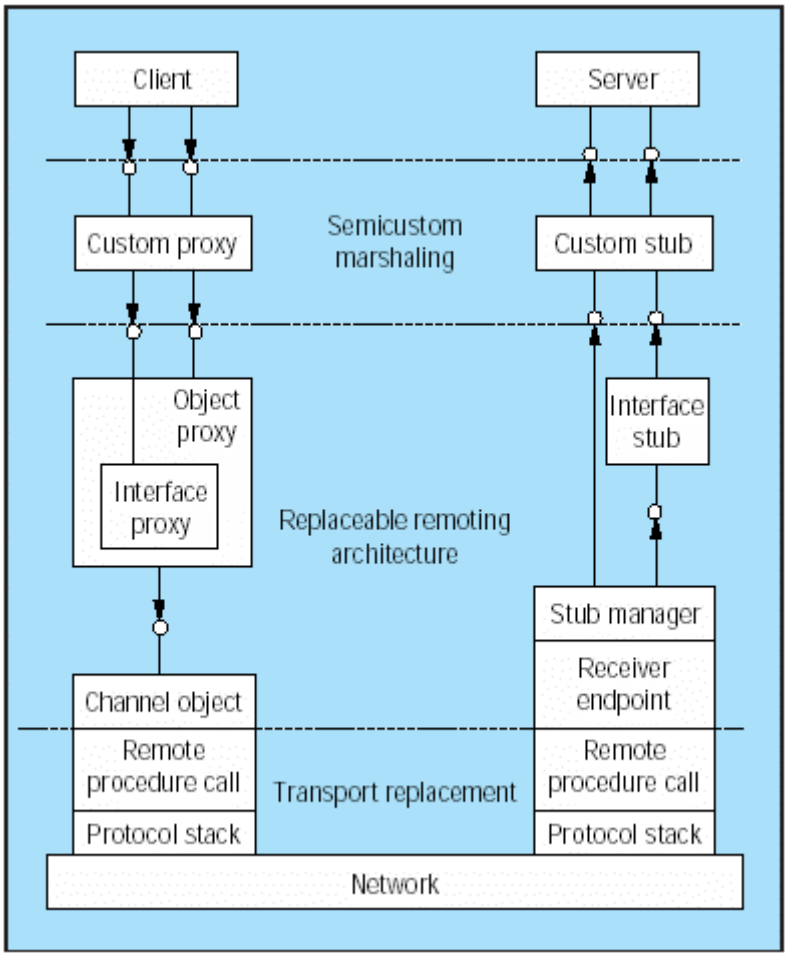


圖十五：位置透明化。

(2) **延伸性**

通常 COM 的應用程式利用標準化的包裝和傳送，但是基於種種原因，應用程式想要訂作自己的連線，例如客戶端也許需要快取唯讀資料夾以加快存取速度，或是整合 DCOM 協力廠商的加密配件或容錯機制，或是分散式共享元件也許希望將資料一致性隱藏在 proxy 之內，種種諸如此類

急需延伸性的應用，在遠端架構上，COM 所提供的延伸性可分成 3 類：below、above 及 within。第一種延伸 COM 在 RPC 層下面。主要的好處在於完全透明於標準化的遠端架構；壞處在於只能應用於傳輸層的改變。目前 DCOM 可以在 TCP、UDP、IPX 或 NetBios 上執行，使用者只須更改 HKEY-LOCAL-MACHINE\Software\Microsoft\Rpc。為了達到其他兩種延伸性，COM 支援訂作的包裝機制。藉著實作 IMarshal 介面，一個伺服物件可以知道要用訂裝的機制。對於第二種延伸，採取的是在標準遠端架構與應用程式間塞入一個處理層。通常在這一層做半包裝的處理。由於有額外的處理，所以需要訂製的 proxy 及 stub。可延伸的部分如下圖十六所示。第三種是最常用的一種，通常應用程式只須少部分訂作的物件，而絕大部分還是重用標準元件。這個延伸以目前的 COM 來說很難達成。利用新的元件化架構:COMERA(COM Extensible Remote Architecture)可以用來達到二元軟體重用的目的。



圖十六：對 DCOM 做延伸

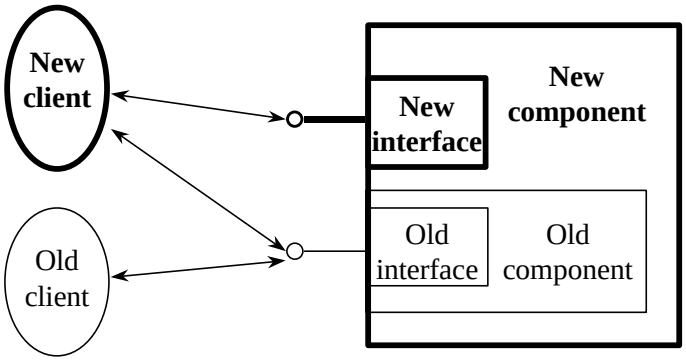
(3) 間接存取

很多軟體的問題可以用間接存取來解決，支援間接存取也可提供延伸性。在傳統程式內提供間接存取也許會造成限制，但是 COM 卻可利用這功能來達到即時軟體更新及負載平衡，同時可以考慮容錯及物件遷移的現象。第一種間接存取發生在當客戶端請求伺服元件啟動時，是藉著它的 CLSID。因為對應這 CLSID 的執行檔是由登錄關鍵字 TreatAs 及 Local Servers 所決定。所以，相同的客戶也許會藉著相同的 CLSID 得到不同的實現，只要關鍵字內容改變了，這樣就可作即時軟體更新。而伺服器可將從別台機器得來的物件參照傳給客戶以達負載平衡。第二個間接存取是發生在介面的 proxy 和 stub。藉著控制登錄(1)IID 對應 CLSID(2)CLSID 對應的檔名，應用程式可以選擇重用或是使用新的。第三個間接存取是發生在 RPC 層所提供之有限制形式的間接存取。假如伺服器的 IP 位址起死回生後到另一台機器，則原來的連結便會斷掉，而 RPC 層便負責將連結轉向至新機器上，下一代 COM+ 將提供 interceptor 之機制來支援此類機制。訂作包裝可以提供間接存取的終極方法呼叫的。基本上，整個遠端架構可視為內建的非直接架構。在比較高層，訂作的 proxy 可以執行客戶端資料相依的檢查，在比較低層，訂作的通道可以動態決定要以那一條實體連線來傳送資料，這個就是客戶端透明的物件遷移和容錯機制的基礎。以上的方法提供間接存取邏輯給 proxy 和通道物件內。

(4) 更新版本

COM 的方法是基於下列三個需求。首先，任何一個 IID 必須是不變的。第二是使用相同的 CLSID 必須支援已存在的介面。最後，任何客戶必須利用詢問介面的方法與伺服器交談。如此才能允許客戶端與伺服器端獨立發展軟體。假設在特定的機器上，伺服器軟體在客戶端軟體更新之前

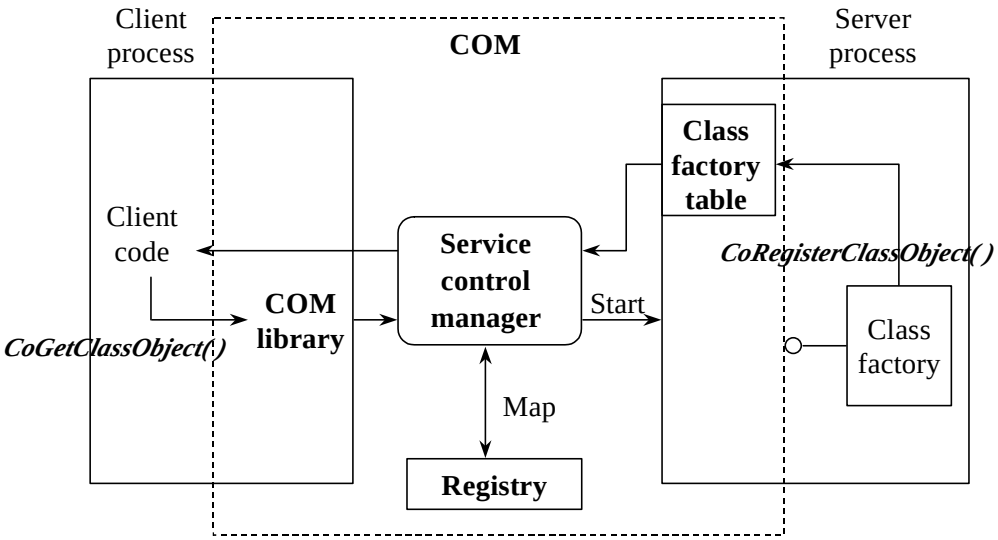
更新，因為新的伺服器支援所有舊有的介面，舊客戶仍然可以保有所有的指標並正常工作。當客戶軟體更新時，新客戶將會詢問新的介面 ID 來享受新機能。相反地假如客戶端軟體先更新，則新客戶將試著詢問舊伺服器而得到失敗的結果。這個程序將使新客戶以舊有的機能來處理這個失誤，但不會使它當掉。圖十七表示更新版本處理的架構。



圖十七：更新版本的處理。

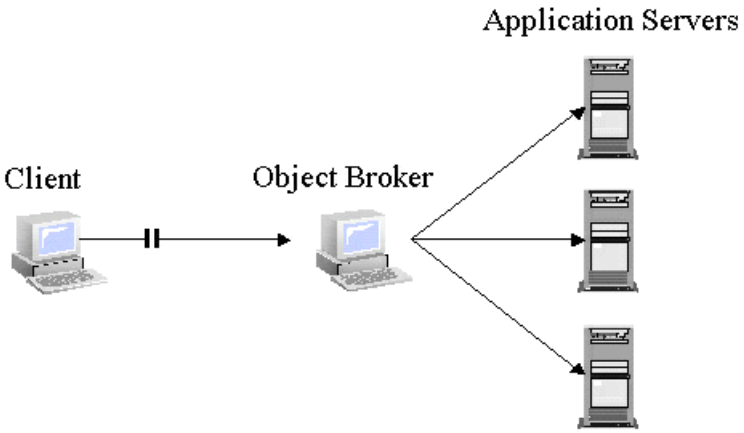
(5) 伺服生命週期的管理

COM 支援各種形態的管理。基本上，被啟動的伺服器產生一個類別工廠來存放所有支援的 CLSIDs 如圖十八。物件實體隨著客戶端的請求而產生，參照計數是用來管理伺服的生命，伺服器在傳出一個參照指標後便將其加 1，而在使用完後便減 1。當減至 0 時，伺服器便知道它不再服務任何客戶，而可以清除掉。為了避免不正常的結束，COM 提供了自動 pinging 的機制，在物件參照解開後，客戶端 ping 碼開始送出週期性的心跳信號給伺服器。當客戶端停止服務時，它也停止送出心跳，在沒接到一定的心跳後，伺服器便會自動切斷連線。



圖十八：伺服物件的管理。

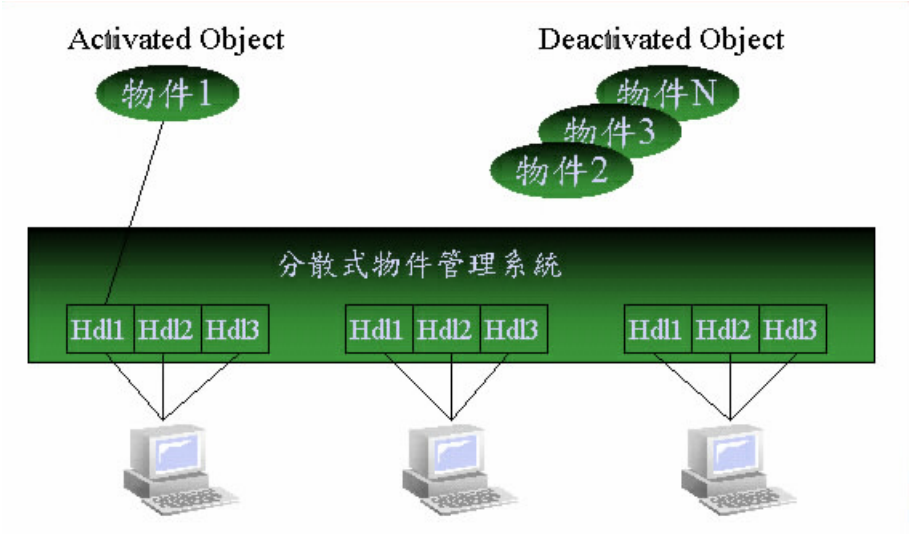
在網路的運算環境下，若所有的服務請求都送給某個特定的伺服器，那麼當客戶端的連線非常多的時候，伺服器的負載就會非常大，可能會超過伺服器的能力而造成系統無法正常運作。如果我們可以設置許多台伺服器，理論上我們可以將服務請求適度的分配給各個伺服器去處理。這個問題的解決方式之一為使用多層式分散式物件技術。若我們在伺服器與客戶端之間加上一層物件中介者層(Object Broker)**，則當有服務請求的時候，客戶端是先將服務請求送給物件中介者。當物件中介者收到這個服務請求之後，它會依照能夠提供此服務的伺服器的負載大小來選定一個適當的伺服器來提供服務。因此，物件中介者就必須要定時偵測伺服器的狀態，才不會將客戶端的服務請求送給無效的伺服器。當伺服器發生錯誤的時候，物件中介者就必須把連到該伺服器的客戶端服務請求轉送給其他的伺服器來完成，運作過程如下圖十九所示。**



圖十九，三層式的架構

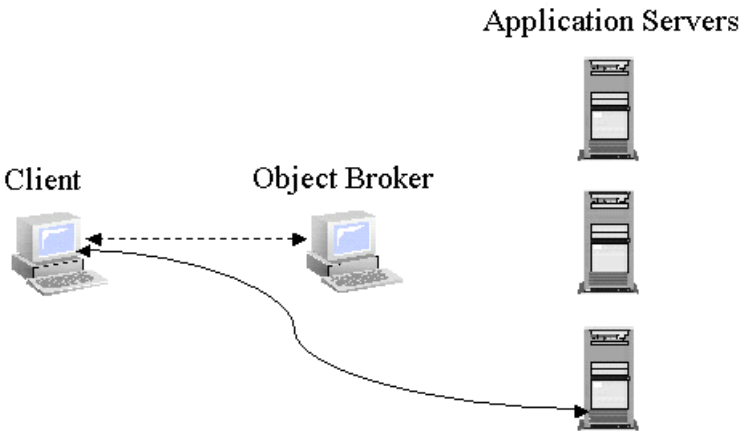
由各種研究發現，這種解決方式對於數量不是很大量的客戶端連線的時候是個可行的方案。但是，當客戶端的連線數量非常多的時候，這個時候反而物件中介者是整個系統的瓶頸。只要物件中介者發生錯誤的時候，整個系統就會停擺，無法正常運作，即使所有的伺服器都還是正常的。因此，對於物件中介者的要求是，即使一台物件中介者失效也不應該造成整個系統失效。解決的方法之一就是對物件中介者也進行叢集(Clustering)**。當某台物件中介者發生錯誤的時候，就由另外一台物件中介者起來取而代之。如此，整個系統就不會存在有一個單一的臨界點**(critical point)**讓系統不穩定。整個系統的容錯機制就可以大大的提昇。此外，由於所有的服務請求都適當的分配到不同的機器上面，所以系統效率也會跟著提高。**

客戶端使用物件管理系統的分散式物件並沒有擁有伺服器物件的直接參考(Reference or Handle)**。客戶端事實上式保有一個由物件管理系統所管理的參考，如下圖二十所示。這層間接層就是讓客戶端完全不必知道某個物件是否是運作中**(active)**或暫停中。**



圖二十：使用分散式物件管理系統來管理物件

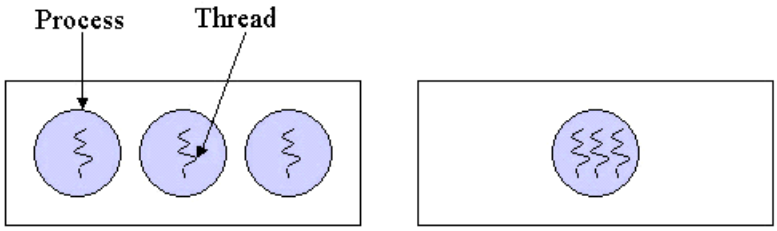
這種方式有許多好處，不過每個客戶端的連線都需要被分散式物件管理系統所管理。如果，能夠讓分散式物件管理系統也可以分散開來，那系統的效率以及可靠度才會提昇。否則，分散式物件管理系統就會是系統的瓶頸，因為對此管理系統來說就是事必親恭。有一種解決方式是讓物件中介者管理所有伺服器所能提拱的服務，但是物件中介者只提供諮詢的角色。也就是說，當客戶端需要服務的時候，它是向物件中介者詢問有哪個伺服器可以提供服務，然後物件中介者送回給客戶端伺服器的位置。接著，客戶端就直接連向伺服器，然後彼此溝通，如圖二十一所示。



圖二十一：透過物件中介者與伺服器直接溝通

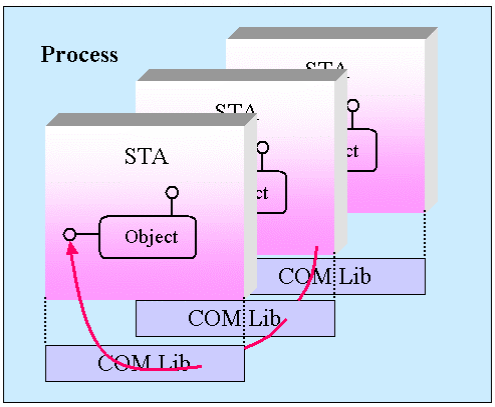
至於物件中介者如何管理伺服器的資料，可以由伺服器啟動時就像物件中介者註冊，而物件中介者每隔一段時間就偵測一次伺服器的狀態，以讓自己維護的資料與伺服器的狀態同步。

由於物件中介者或者伺服器經常需要處理客戶端的服務請求，因此處理效率也是必須要考慮的因素之一。圖二十二中表示兩種不同的執行模式。由於 CPU 在不同的行程之間切換所需要的切換時間，對行程來說遠大於對執行緒。此外，行程除了切換時間以外還有其他比執行緒還多的額外負擔。因此，系統不應該讓一個客戶端連線請求就產生一個行程來處理它，而是應該使用執行緒來處理。除此之外，由於不同的行程存在於不同的位址空間，因此若要做到資料共享的話勢必要有許多額外的步驟才能達成，這些都將造成額外的負擔。在分散式的運算環境中，資料共享是經常遭遇到狀況。例如，物件中介者可能擁有一個關於各個伺服器狀態的資料，而每個客戶端的連線請求都會需要使用到這個資料。很明顯的，如果是使用行程中單一執行緒的方式效果就不佳。

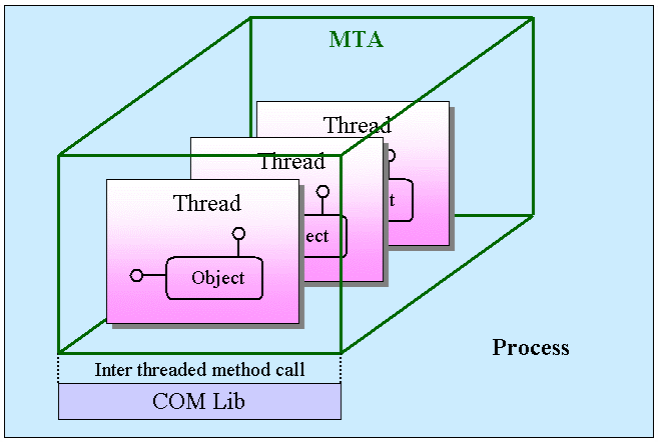


圖二十二：不同的執行模式

DCOM 提供了兩種多執行緒的執行模式，分別是 Single-Threaded Apartment (STA)以及 Multiple-Threaded Apartment (MTA)。一個行程中可以包含有數個 STA，不過卻只能有一個 MTA。每個 STA 之內只有一個 thread（如圖二十三），而每個 MTA 裡面可以一個以上的 threads（如圖二十四）。



圖二十三：STA 模式



圖二十四：MTA 模式

只要是跨越 apartment 邊界的話就需要自己處理資料同步以及資料打包與解封包的動作。由於 STA 之間的呼叫會跨越 apartment 邊界，所以需要做資料打包與解封包，不過 STA 裡面的資料會自動同步。因為 STA 使用了一個隱藏的視窗來處理訊息。由於訊息是存放在隱藏視窗的訊息佇列中，因此 STA 之間的動作會自動同步。至於，MTA 就不需要做打包的動作，不過卻得自己處理同步的問題。

我們實作了類似物件中介者以及伺服器功能的程式用來測試系統。這兩種程式都是使用一個 MTA 與數個 STA。而客戶端是哪種模式就比較不重要。不過基於撰寫方便，我們選擇使用 STA 模式並且利用微軟公司的 ATL 來幫助撰寫程式碼。中介者的 MTA 除了有監控伺服器的執行緒以外，也可以視

需要而產生工作執行緒來完成所交付的任務。而伺服器的 MTA 則比較像是一個 thread pool，裡面有著一些已經等待任務一到就可以馬上執行的執行緒，等到執行結束以後就把執行緒歸還給伺服器。若 thread pool 裡面的執行緒不夠的話，則伺服器可以再產生新的執行緒以滿足需求。物件中介者的輸出片段列在下圖二十五。其中，最左邊為時間，而 COMPUTER1 等等為伺服器名稱，此名稱上面大括號所包住的為伺服器所提供的物件的 GUID。

```
19:20:33    {993A3C62-0072-11D5-BDF4-0080C86F9FE7}
        COMPUTER1

19:20:35    {993A3C62-0072-11D5-BDF4-0080C86F9FE7}
        COMPUTER1
        COMPUTER3

19:20:35    {993A3C62-0072-11D5-BDF4-0080C86F9FE7}
        COMPUTER1
        COMPUTER2
        COMPUTER3

19:20:38::00ms - pinging COMPUTER1, loading = 0, napping = 0

19:20:40::520ms - pinging COMPUTER2, loading = 0, napping = 0

19:20:57::820ms - pinging COMPUTER3, loading = 1, napping = 0

19:21:47::810ms - pinging COMPUTER2, loading = 0, napping = 0

19:21:47::810ms - pinging COMPUTER1, loading = 0, napping = 0

19:21:49::510ms - pinging COMPUTER3, loading = 4, napping = 0
…
```

圖二十五：中介者的執行結果輸出片段

而每個伺服器有自己的輸出結果，我們在此處只列出其中一個的片段在下圖二十六中。由於這個版本中，只要客戶端的服務要求送到伺服器，伺服器就會計算自己的負載程度是多少, 由這個片段中我們可以看到這樣的結果。

```
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 10 to it.
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 10 to it.
Call Broker: COMPUTER1
        CoCreateInstanceEx() return successfully
        RegNameAndService() returns successfully
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 0 to it.
19:13:20::350 ms - Client (COMPUTER1) calls my Service1(). Now loading = 1
```

```
19:13:20::840 ms - Client (COMPUTER1) calls my Service1(). Now loading = 2
19:13:20::840 ms - Client (COMPUTER1) calls my Service1(). Now loading = 3
19:13:20::900 ms - Client (COMPUTER1) calls my Service1(). Now loading = 4
19:13:20::900 ms - Client (COMPUTER1) calls my Service1(). Now loading = 5
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 5 to it.
Call Broker: COMPUTER1
        CoCreateInstanceEx() return successfully
        RegNameAndService() returns successfully
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 0 to it.
19:20:42::60 ms - Client (COMPUTER1) calls my Service1(). Now loading = 1
19:20:42::550 ms - Client (COMPUTER1) calls my Service1(). Now loading = 1
19:20:42::610 ms - Client (COMPUTER1) calls my Service1(). Now loading = 3
19:20:42::660 ms - Client (COMPUTER1) calls my Service1(). Now loading = 4
19:20:42::720 ms - Client (COMPUTER1) calls my Service1(). Now loading = 5
Broker (COMPUTER1) calls my GetLoading(). Returns loading = 0 to it.
…
```

圖二十六：伺服器的輸出片段

由於伺服器程式與中介者程式可以在同一部電腦上面執行，所以上面的輸出片段可以看出 COMPUTER1 **除了拿來當伺服器也拿來當中介者。因此，它計算負載的時間會隨著其它行程的負載增加而變慢。因此，大部分的電腦不是單純做伺服器就是做中介者，否則計算負載的方式就得計算整個電腦內部的所有運算。**

在分析了程式的輸出結果後，我們發現利用此方式建構的系統效率與可靠度有顯著的提昇。客戶端會透過物件中介者而連上適當的伺服器要求服務，客戶端不會因為連上無效的伺服器而造成錯誤。當特定伺服器因為某種緣故而無法使用的時候，客戶端的請求不會被送到該伺服器，因為中介者會知道該伺服器已經無法使用而將請求送給有效的伺服器。而伺服器也不會因為所有的服務請求都導向它而造成系統無法運作，因為中介者保有一份伺服器的資料可以決定比較適當的伺服器來提供服務。使用 MTA **的執行模式與** thread pool **可以使得程式的資料共享變的更加容易，程式的效率提昇後，系統的整體效率也跟著提昇了。**

五、結論：

在此子計畫的執行過程中，我們仔細研讀並比較了 COM/DCOM、CORBA 等分散式物件技術的規格以及其運作方式，從中獲得了各項技術的關鍵知識。另外，我們依據 DCOM 技術的規格實際撰寫了一些應用程式，並觀察程式的執行流程，以深入瞭解 DCOM 的運作機制，配合其它子計劃著手實作初步的網路醫學資訊系統。我們也研究了在分散式系統中提昇系統效率、可靠度、可擴充性、容錯機制的方法。在研究如何將多層式架構實際應用在系統中時，對於多層式架構的可能問題做改善以提昇系統可靠度及效率。此外，我們也研究在分散式物件系統下，不同的程式執行模式對系統的影響，因而選擇使用 MTA 與 STA 配合的方式。

本計畫建立的遠距醫療資訊系統，強調系統服務提供的即時性與穩定性。病人可以透過網路直接與醫師進行互動，而身處各地的醫師們則直接在線上進行會診，無需費心於網路醫學資訊系統本身可能出錯所造成之影響。在即時性與穩定性上，我們使用叢集的多層式系統架構搭配容錯的機制以及多執行緒的程式執行模式來實現，以提供高效能的整合性醫療服務。而系統的可擴充機制，是針對未來的發展而設計。這些研究與實作成果將為發展其他網際網路上的資訊系統服務奠定了穩固的基礎。

分散式計算已經成為主流，加上物件導向程式也已成為發展可重用性軟體的主流，分散式物件技術結合這兩大趨勢而漸漸受歡迎。軟體系統建立成分散式物件可讓應用程式彼此間分享許多資源與共同的目標。藉由分散式物件技術的廣泛應用，未來的軟體發展將有非常快速的進展。掌握了分散式物件這個熱門的領域，就可以在網路服務佔有一席之地。此計畫的實作縮短了學術研究和產業界在實作能力的差異，參與之工作人員除可獲得物件導向程式設計能力、分散式物件技術的實作能力以外，更對容錯應用有更深入的了解。

六、參考文獻：

[1] Dale Rogerson, "*Inside COM*", Microsoft Press, 1996.

[2] Don Box, "*Essential COM*", Addison Wesley, 1998.

[3] Stanley Lippman, "*Inside the C++ Object Model*", Addison Wesley, 1996.

[4] Microsoft Corporation and Digital Equipment Corp., "*The Component Object Model Specification*", http://premium.microsoft.com/isapi/devonly/prodinfo/msdnprod/msdnlib.idc?theURL=/msdn/library/specs/tech1/d1/s1d137.htm

[5] N. Brown and C. Kindel, "*Distributed Component Object Model Protocol – DCOM/1.0*", Internet Draft, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt, Nov. 1996.

[6] OSF DCE RPC Specification, http://www.osf.org/mall/dce/free_dce.htm, 1994.

[7] COM/DCOM Resources, http://www.research.att.com/~ymwang/resources/resources.htm.

[8] "ActiveX Core Technologies Description", http://www.activex.org/announce/ActiveX_Core_Technologies.htm.

[9] Y. M. Wang, "*Introduction to COM/DCOM*", tutorial slides, http://www.research.att.com/~ymwang/slides/COMHTML/ppframe.htm, 1997

[10] "The Ensemble Distributed Communication System", http://simon.cs.cornell.edu/Info/Projects/Ensemble/index.html.

[11] "*Clustering Solutions for Windows NT*", Windows NT Magazine, pp. 54-95, June 1997.

[12] Y. Huang and C. Kintala, "*Software implemented fault tolerance: Technologies and experience*", in Proc. IEEE Fault-Tolerant Computing Symp., pp. 2-9, June 1993.

[13] D. Chappell, "*Understanding ActiveX and OLE*", Microsoft Press, 1996.

[14] "*DCOM Technical Overview*", http://www.microsoft.com/windows/common/pdcwp.htm.

[15] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y. M. Wang, "*ONE-IP: Techniques for Hosting a Service on a Cluster of Machines*", *Proc. 6th WWW Conference*, pp. 735-743, April 1997.

[16] Jason Pritchard, "*COM and CORBA side by side*", Addison Wesley, 1999.

[17] K. P. Birman, "*Building Secure and Reliable Network Applications*", Greenwich, CT: Manning Publications Co., 1996.

[18] K. Brockschmidt, "*Inside OLE*", Microsoft Press, 1993.

[19] D. Box, "*Q&A ActiveX/COM*", Microsoft Systems Journal, pp. 93-105, March 1997.

[20] Y. M. Wang, Y. Huang, and W. K. Fuchs, "*Progressive retry for software error recovery in distributed systems*", *in* Proc. IEEE Fault-Tolerant Computing Symp., pp. 138-144, June 1993.

[21] P. E. Ammann and J. C. Knight, "*Data diversity: An approach to software fault-tolerance*," IEEE Trans. Computers, Vol. 37, No. 4, pp. 418-425, Apr. 1988.

[22] A. Avizienis, "*The N-version approach to fault-tolerant software*", IEEE Trans. Software Eng., Vol. SE-11, No. 12, pp. 1491-1501, Dec. 1985.

[23] B. Randell, "*System structure for software fault tolerance*", IEEE Trans. Software Eng., Vol. SE-1, No. 2, pp. 220-231, June 1975.

[24] R. Grimes, "*Professional DCOM Programming*", Wrox Press, 1997.

[25] W. Vogels, "*A programming environment for building cluster-aware DCOM applications*", private communication, June 1997.

[26] J. Siegel, "*COBRA Fundamentals and Programming*", John Wiley & Sons, 1996.

[27] M. T. Ozsu and P. Valduriez, "*Principles of Distributed Database Systems*", Prentice Hall, 1991.

[28] E. Seligman and A. Beguelin, "*High-level fault tolerance in distributed programs.*" Tech. Rep. No. CMU-CS-94-223, Dept. of Computer Science, Carnegie Mellon University, 1994

[29] A. S. Tanenbaum, A. S. Woodhull, "*Operating Systems: Design and Implementation*", Prentice Hall, 1997.

[30] G. Chen, "*Distributed transaction processing standards and their applications*," Computer Standards and Interfaces, No. 17, pp. 363-373, 1995.

[31] T. Berners-Lee, R. Fielding, H. Frystyk, "*Hypertext Transfer Protocol*", HTTP Working Group Informational document, RFC 1945, May 1996. http://www.ics.uci.edu/pub/ietf/http/rfc1945.ps.gz

[32] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Bernes-Lee, "*HyperText Transfer Protocol – HTTP/1.1*", HTTP Working Group Proposed Standard, RFC 2068, Jan. 1997. http://www.ics.uci.edu/pub/ietf/http/rfc2068.ps.gz

[33] D. M. Kristol and L. Montulli, "HTTP State Management Mechanism", *HTTP Working Group, Proposed Standard*, RFC 2109, Feb. 1997. http://www.ics.uci.edu/pub/ietf/http/rfc2109.txt

[34] B. Narendran, S. Rangarajan and S. Yajnik, "Data Distribution Algorithms for Fault-Tolerant Load Balanced Web Access", in *Proc. of the IEEE Symposium on Reliable Distributed Systems*, October 1997.

[35] S. Rangarajan, S. Yajnik and P. Jalote, "*WCP – A tool for consistent on-line update of documents in a WWW server*", http://www.bell-labs.com/~shalini/www7conf/journal.html.

[36] M. Sullivan and R. Chillarege, "*Software defects and their impact on system availability – A study of field failures in operating systems*", in Proc. IEEE Fault-Tolerant Computing Symp.,

pp. 2-9, 1991.

[37] J. Gray and D P. Siewiorek, "*High-availability computer systems*" IEEE Comput. Mag., pp. 39-48, Sept. 1991.

[38] B. Randell, "*System structure for software fault tolerance*", IEEE Trans. Software Eng., Vol. SE-1, No. 2, pp. 220-232, June 1975.

[39] E. Adams, "Optimizing preventive service of software products", *IBM J. R&D*, No.1, pp. 2-14, Jan. 1984.

[40] Y. Huang and C. Kintala, "A software fault tolerance platform", in Practical Reusable Software, Ed. B. Krishnamurthy, pp. 223-245, John Wiley & Sons, 1995.

[41] R. E. Strom, D. F. Bacon, and S. A. Yemini, "Volatile logging in n-fault-tolerant distributed systems", in *Proc. IEEE Fault-Tolerant Computing Symp.*, pp. 44-49, 1988

[42] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems", ACM Trans. Comput. Syst., Vol. 3, No. 3, pp. 204-226, Aug. 1985.

[43] Y. M. Wang and W. K. Fuchs, "Lazy checkpoint coordination for bounding rollback propagation", in Proc. IEEE Symp. Reliable Distributed Syst., pp. 78-85, Oct. 1993.

[44] G. Fowler, Y. Huang, D. Korn, and H. Rao, "A user-level replicated file system", in Proc. Summer '93 USENIX, pp. 279-290, June 1993.

[45] Yi-Min Wang, Pi-Yu Emerald Chung, "Customization of distributed systems using COM", IEEE Concurrency, Jul-Sep 1998.