

國立交通大學資訊工程學系

專題研究報告書

研究題目

同儕網路上的影音串流群播服務

Streaming Multicast Service over P2P Network

組員：

9117006 黃得源

9117016 蔡佩娟

9117019 郭芳伯

9117106 黃筠茹

指導教授：邵家健教授

中華民國 94 年 12 月 29 日

# 目錄

---

---

|                                       |       |
|---------------------------------------|-------|
| ➤ 專題動機.....                           | P. 03 |
| ➤ 文獻探討.....                           | P. 03 |
| v Infrastructure .....                | P. 03 |
| v Application Layer Multicast.....    | P. 04 |
| ➤ 研究方法.....                           | P. 09 |
| v 選擇 P2P Overlay Infrastructure ..... | P. 09 |
| v ALM.系統設計.....                       | P. 11 |
| * SRS.....                            | P. 11 |
| * SDS.....                            | P. 12 |
| * Low Level Design.....               | P. 16 |
| ● Join Algorithm P.17                 |       |
| ● Depart Algorithm P.17               |       |
| ● Split Algorithm P.20                |       |
| ● Merge Algorithm P.26                |       |
| v Media Framework.....                | P. 31 |
| v Communication Over JXTA.....        | P. 31 |
| v UI Design .....                     | P. 32 |
| ➤ 計畫成果.....                           | P. 32 |
| v Customized JxtaBidiPipe.....        | P. 32 |

v **Our Implementation P.32**

v **Demo**

- **結語與未來方向..... P. 21**
- **參考資料..... P. 21**
- **附錄一..... P. 24**

## 一、 專題動機

Skype網路電話服務的成功，讓我們在電信服務的使用習慣上頭有了重大的轉變。便利且便宜的網路電信服務讓人與人之間的距離瞬間縮小許多，越來越可以不受限制的進行交流。Skype的成功是因為其價格低廉且音質良好，而促成這些優點的是他底層的P2P Overlay Network。P2P Overlay Network不需要伺服器，所以不需要存在有一個團隊在背後維護伺服器。對於使用者而言，這代表了不需要付費給維護伺服器的團隊。對於整體架構而言，則代表了不會因為伺服器的損壞而導致架構的不能運行。

於是，我們想要嘗試在P2P Overlay Network上頭架構Video Streaming Multicast Service。讓使用者可以透過我們的架構，享受影像電話般的對話品質。同時，也可以利用我們所實做的群播服務，讓使用者及時的和多人進行交談，享用更多元化的網路服務。

## 二、 文獻探討

### I. Infrastructures

#### 1. JXTA

JXTA 是一個專為同儕網路(P2P)的處理而設計的開放性網路平台。其目標是開發基本元件和服務，幫助發展以群組(Peer Group)為主的應用。

“JXTA” 這個詞是來自 juxtapose，意為 side by side。這個名字的意義在於，相對於現今傳統的分散式運算架構，如：伺服器-終端架構(client-server)或以網頁為基礎的架構(web-based computing)，P2P 是一個並行的概念。JXTA 提供了一系列基本 open protocols 和一個發展同儕網路應用的平台。JXTA 的 protocols 為一個 peer 制定了以下的標準：

- Discover each other
- Self-organize into peer groups
- Advertise and discover network services
- Communicate with each other
- Monitor each other

Figure6 為 JXTA 的大致架構圖：

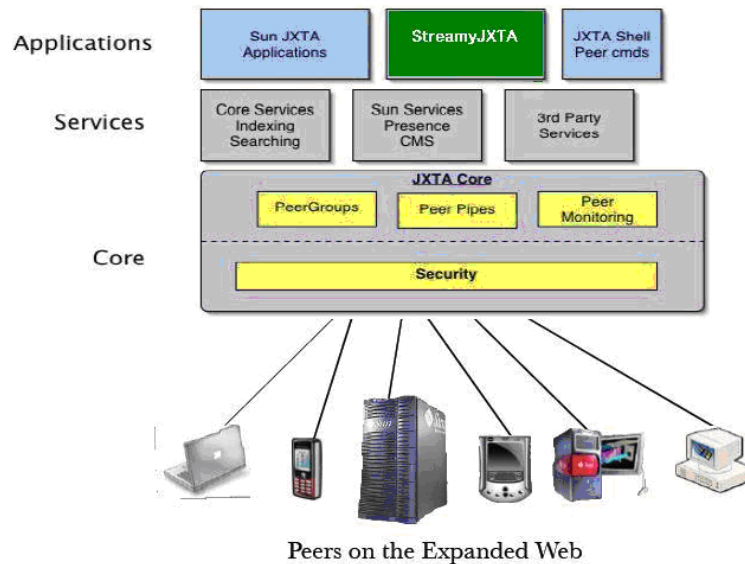


Figure6

JXTA 架構圖

## II. Application Layer Multicast

### 1. Zigzag 演算法簡介

事實上目前的 Internet 並不普遍支援網路上的 multicast protocol，而 P2P 的應用卻開始產生大量的 Streaming multicast 傳輸需求。在這個 ALM 裡，透過 Zigzag 的方法，我們可以將 peers 分成 clusters，並將 clusters 組織成群播樹狀結構(multicast tree hierarchy)，不僅易於管理，也更有效率的傳輸資訊。

在 Zigzag 演算法中，multicast tree 的高度是使用者數(clients)的對數( $\log n$ )，而每個 node 需要傳送的下端 clients 數量也限制在常數內。透過這樣的方式，可以減少從資訊源(streaming source)到使用者端需要經過的傳輸點(relay node)個數，同時也避免了網路傳輸頻寬的上限瓶頸。如此便可控制端點到端點之間的時間差在極小狀態。一個有效的 protocol 如 Zigzag，可以有效的在動態網路上維持這個 tree 結構。在最差的情況下，Zigzag 的 time complexity 也是  $O(n)$ 。尤其在一定數量的使用者離開系統的時候，Zigzag 也可以很快的在常數時間內在局部做出錯誤處理(failure recovery)，而避免給整個系統帶來負擔。

Figure7 是 Zigzag 所呈現的 multicast tree 架構：(箭頭方向就是 streaming 傳輸的方向)

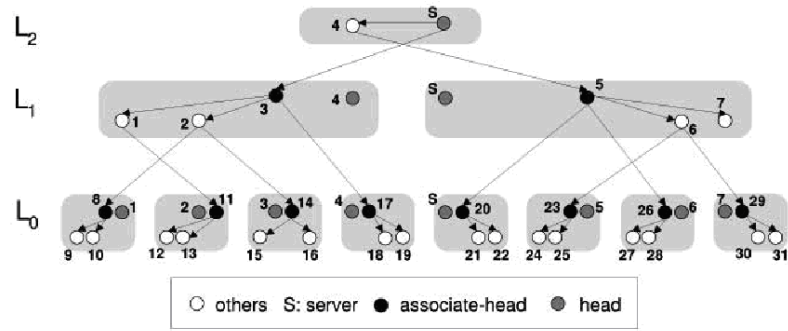


Figure7

Multicast tree atop the administrative organization

## 2. Zigzag 名詞定義

### ➤ Subordinate

在下圖中，灰色的點代表 Head，和 Head 同屬一個 cluster(灰色區塊)的點即為該 Head 的 Subordinate。如下圖所示：4 為 S 之 subordinate，3 為 4 之 subordinate。

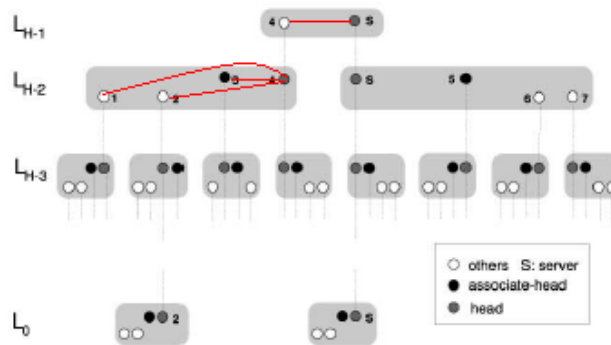


Figure8

Illustration of Subordinate

### ➤ Foreign Head

在 Layer j 中非 Head 的 node 為 Layer j-1 中和自己無關的 cluster 中之非 Head 的 node 的"Foreign Head"。如下圖，node 1 為 node 2、12、13 的 Foreign Head。

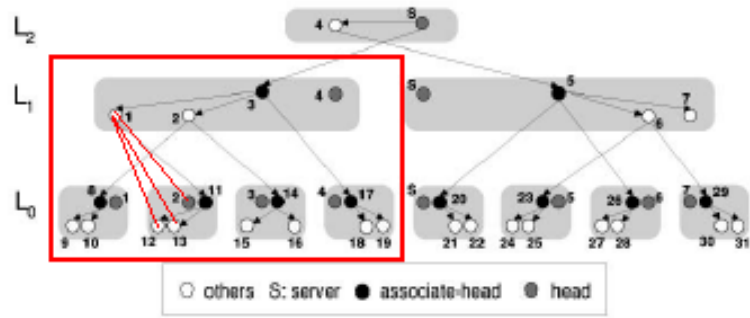


Figure9  
Illustration of Foreign Head

- Foreign Subordinate  
在 layer  $j-1$  的 associate head 為其 Foreign Head 的 Foreign Subordinate，如下圖：2、3、4 為 Node 1 的 Foreign Subordinate。



Figure10  
Illustration of Foreign Subordinate

- Foreign Cluster  
在 Layer  $j$  中，如果其 Head 在 Layer  $j+1$  屬於同一個 cluster，則這些 cluster 互為彼此的 Foreign Cluster，如下圖所示，以紅色線條相連的 cluster，互為 Foreign Cluster。

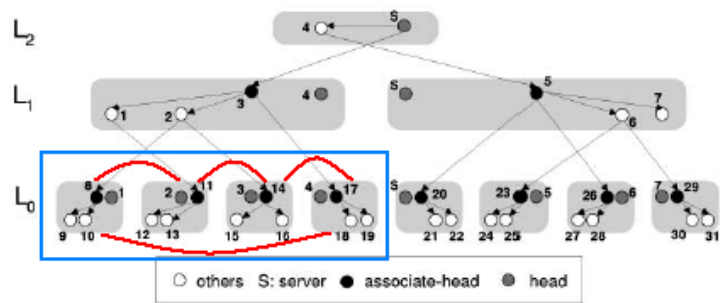


Figure 11  
Illustration of Foreign Cluster

➤ Super Cluster

Layer  $j$  某一 cluster 的 Head 在 Layer  $j+1$  所屬的 cluster 為前述 cluster 的 Super cluster，如下圖所示，兩個以紅色方框框起的 cluster，上方是下方的 super cluster。

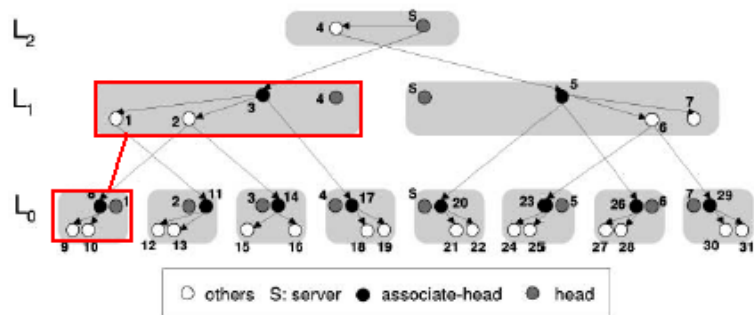


Figure 12  
Illustration of Super Cluster

3. ALM Tree 的基本架構

- ✓ Layer 0 包含了所有的 peers
- ✓ 一個 cluster 由  $[k, 3k]$  個 peers 所組成。這些 peer 中，會挑選其中的兩個 peer 為 Head 和 Associate Head。
- ✓ 被挑選成為 Head 的 peer 將會成為下一 Layer 的 member。
- ✓ 在整棵樹的最高層，只有一個 cluster，這個 cluster 由  $[2, 3k]$  個 node 所組成。
- ✓ 在整棵樹的最高層，source peer 同時會是 Head 及 Associate Head。

#### 4. ALM Tree 的連接規則

##### ✓ Rule 1

當一個 peer 不在他所處的最高層的時候，既不會有 in link 也不會有 out link。

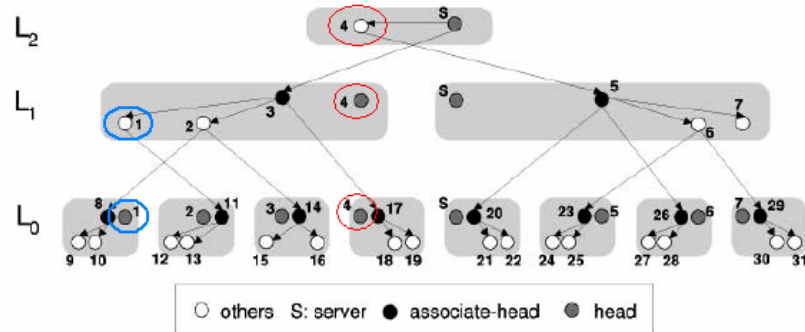


Figure13

Illustration of Rule 1

##### ✓ Rule 2

一個 cluster 中，既非 Head 也非 Associate Head 的 node 只能從 Associate Head 得到 streaming data。

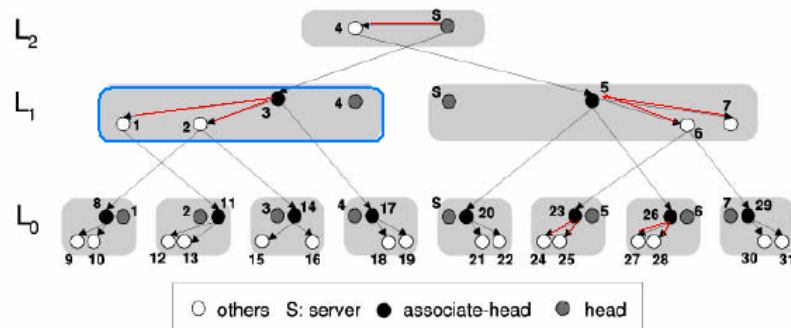


Figure14

Illustration of Rule 2

##### ✓ Rule 3

Cluster 的 Associate Head 必須從他的 Foreign Head 取得 streaming data。

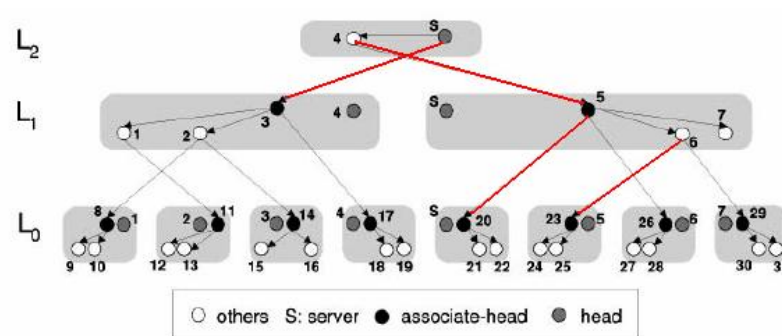


Figure15  
Illustration of Rule 3

### 三、 研究方法

#### I. 選擇 P2P Overlay Infrastructure

既然我們的 Application 是要架構在 P2P Overlay Network 之上，必然需要先找尋適合的 P2P Infrastructure。我們在 structured P2P Infrastructure 中挑出了環形的 Chord 和樹狀的 Tapestry 來比較，至於 unstructured P2P 的部分，則是挑出了 JXTA。

#### ➤ Tapestry、Chord 及 JXTA 的比較

#### ✓ Structured & Unstructured

首先，我們就 Structured 和 Unstructured 的 P2P Infrastructure，來做比較。

當我們稱一個 P2P Overlay Network 為 Unstructured 的架構時，表示在這個 Overlay Network 中，其 Contents/Files 的 placement 和 Overlay 的 topology 是沒有關係的。相對的，當我們稱一個 P2P Overlay Network 為 Structured 的架構的時候，表示其 overlay topology 是被嚴密的控制且其上的 Contents/Files 是被放置在一個特定的位置，同時系統會提供 content 和 location 之間的 mapping 關係(即一般所熟知的 Distributed Hashing Table，簡稱 DHT)，讓 Contents/Files 易於 locate。

|              | 優點  | 缺點  |
|--------------|---|---|
| Structured   | <ul style="list-style-type: none"> <li>* 擴充性(scalability)佳</li> <li>* 只要 Content 存在，保證可以被找到</li> <li>* 已有架構提出支援無線網路的方法</li> </ul> | <ul style="list-style-type: none"> <li>* 實做複雜</li> <li>* 節點流動率高的情況下，難以維護</li> <li>* 目前架構仍在測試階段</li> <li>* 無法以關鍵字搜尋資源(受限於 DHT)</li> </ul>                |
| Unstructured | <ul style="list-style-type: none"> <li>* 實做容易</li> <li>* 可以關鍵字搜尋資源</li> </ul>   | <ul style="list-style-type: none"> <li>* 若無 Server 支援，只能以 Flooding 或 Random Walk 等 Brute Force 來搜尋 Content</li> <li>* Content 即使存在也未必能保證可被找到</li> </ul> |

✓ Structured P2P Overlay：Tapestry & Chord

除了在大項目上做比較之外，我們也從 Structured P2P Overlay 中挑出了 Tapestry 和 Chord 兩個 P2P Infrastructure 來做比較：

|          | 優點  | 缺點   |
|----------|---|--|
| Tapestry | <ul style="list-style-type: none"> <li>* 採用樹狀結構</li> <li>* 擴充性佳</li> <li>* 已提出支援無線網路的架構 (Warp)</li> <li>* 安全性佳</li> </ul>                         | <ul style="list-style-type: none"> <li>* 實做複雜</li> <li>* 鄰居地圖 (neighbor map) 建置不易</li> </ul> |
| Chord    | <ul style="list-style-type: none"> <li>* 工作量負載平均</li> <li>* 路由效率較佳</li> <li>* 已有 SIP 嘗試實做 Chord 的前例</li> <li>* finger table 建置容易，實做較簡單</li> </ul> | <ul style="list-style-type: none"> <li>* 節點的移動能力 (mobility) 差</li> <li>* 安全性稍差</li> </ul>    |

✓ Unstructured P2P Overlay：JXTA

另外，我們也從 Unstructured 的角度切入，挑選 JXTA 此一架構來進行評估：

|      | 優點   | 缺點   |
|------|--|--|
| JXTA | <ul style="list-style-type: none"> <li>* 已有開發平台存在</li> <li>* 文件豐富</li> <li>* 有專門團隊在維護開發環境</li> </ul> | <ul style="list-style-type: none"> <li>* Content 即使存在也未必能保證可被找到</li> <li>* 存在 rendezvous server，並非 pure P2P</li> </ul> |

✓ 底層架構的選擇

一開始我們被 Structure 的 DHT 架構所能提供的保證性吸引，但是經過架設平台等等的過程發現 Tapestry 的結構過於複雜，導致其所開發出來的底層架構不僅維護不易，其本身的 Bug 也相當的多。相對的 JXTA 本身已經是開發得相當完善的平台，使用 JXTA 可以讓我們專注於上層 Application 的開發，而不必分神於底層架構的問題。

最後我們選擇 **JXTA** 作為專題的底層架構。

## II. 實作 ALM—系統設計

### 1. SRS

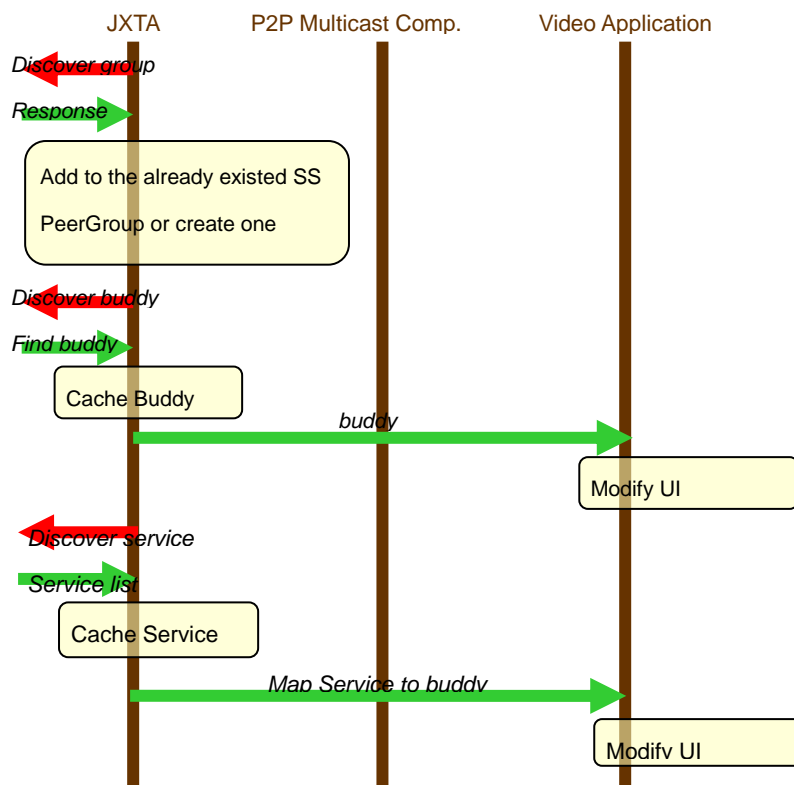
➤ 優先提供的服務

- ✓ Peer 可提供 video streaming service
  - Accept peer request
  - Reject peer request
- ✓ Peer 可接收 video streaming service
- ✓ 可搜尋其他 Peer
- ✓ 提供控制面版供使用者使用
  - Provide service button  
按下後便開始提供 video streaming service
  - Source Peer List  
目前有那些 Peer 提供 service
  - Service button: Indicating buddy is providing service  
按下後便送出 request 給 source peer，要求接收 video streaming service
  - Watching icon: Which peer is watching my service

- 次要提供的服務
  - ✓ Message  
Peer 和 peer 之間的訊息傳遞
  - ✓ Presence  
上線下線的狀態管理
  - ✓ Kick out watching peer  
當身為 service provider 時，該 peer 有權終止他人觀看自己提供的影像

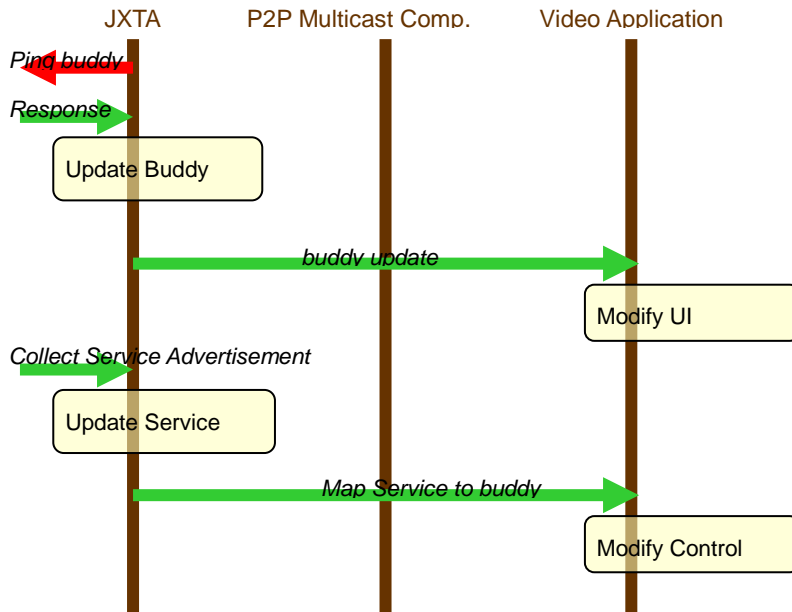
## 2. SDS

### ◆ Application Initialization



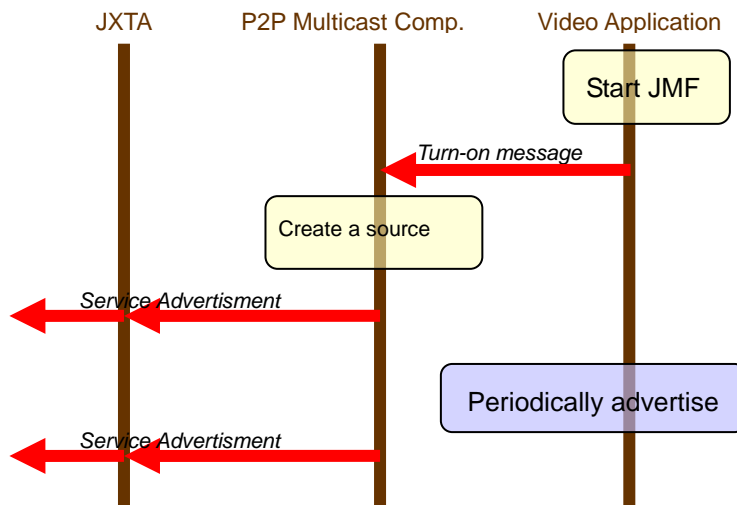
1. 透過JXTA平台搜尋SS PeerGroup並加入，若找不到變建立一個新的SS PeerGroup。
2. 搜尋SS PeerGroup上的其他Buddy，儲存並顯示在UI上。
3. 搜尋SS PeerGroup上 Buddy所提供的 Service，儲存並顯示在UI上。

◆ Periodically Updated



1. 定期確認 Buddy 是否還在 SS PeerGroup，儲存 Update 結果並顯示在 UI 上。
2. 定期搜尋 Buddy 所提供的 Service，儲存並顯示在 UI 上。

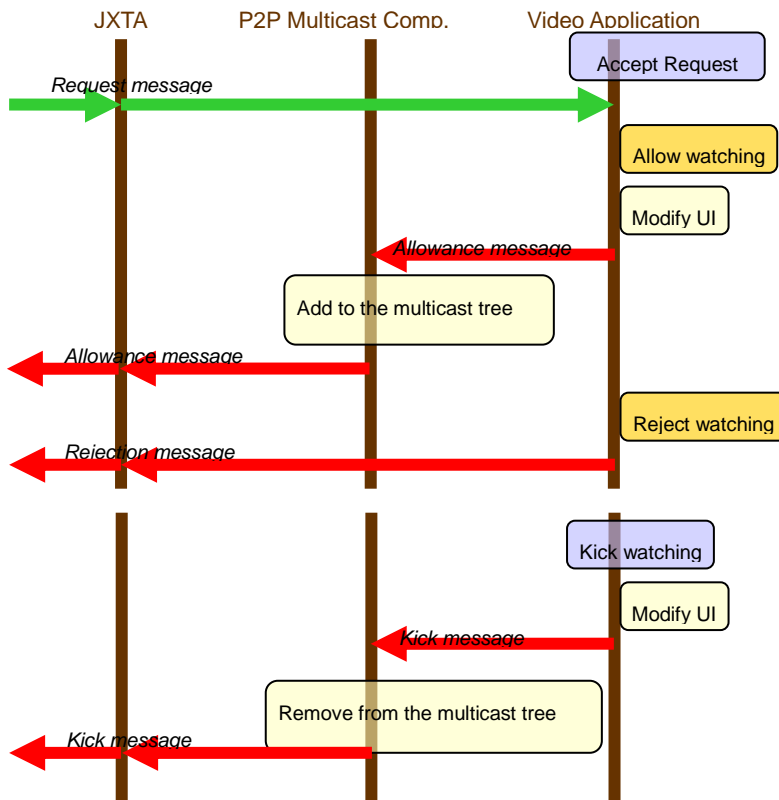
◆ Provide Service



1. 按 provide service button 後，啟動 JMF，並建立一個 multicast tree 的 source，然後透過 JXTA 發出提供 Service 的訊息。
2. 之後定期發出 Service Advertisement，代表持續提供

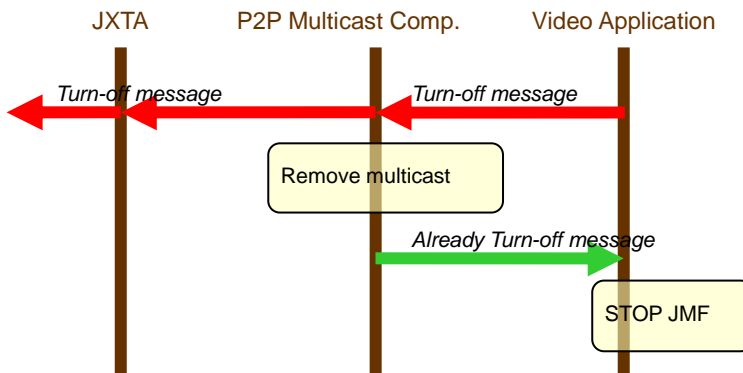
Service，並讓後來加入的 Buddy 也能夠看到。

◆ **Maintain Service : Accept Request and Kick**



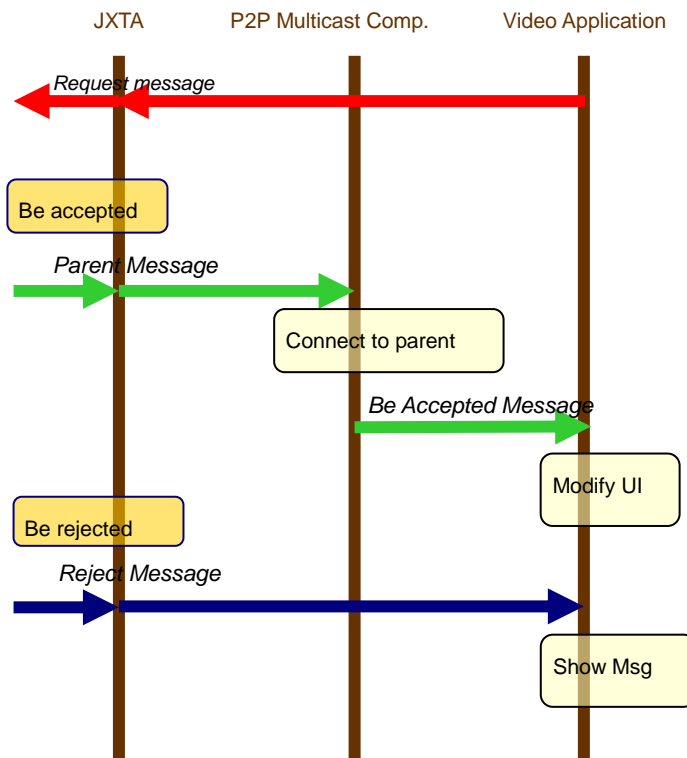
1. 接受到 request service 訊息：  
 接受：顯示在 UI 上，並將其加入到 multicast tree 中，然後傳送接受訊息。  
 拒絕：傳送拒絕訊息。
2. 在 UI 上點選要踢掉的 Buddy，將 Buddy 從 multicast tree 移除後，在將訊息傳送給被踢掉的 buddy。

◆ **Stop Service : Accept Request and Kick**



1. 發出關閉 Service 訊息，通知其他觀看的人離開後，將 multicast tree 結束，並回傳以結束訊息，然後停止 JMF 運作。

◆ Request Service from Others :



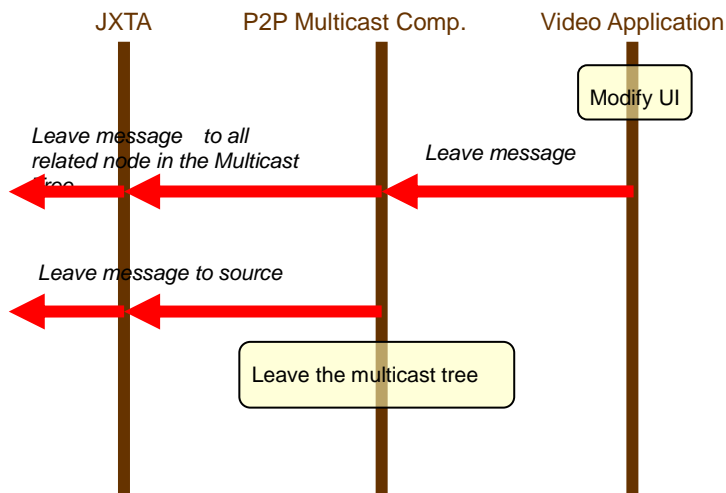
1. 發出觀看他人 Service 請求。

被接受：

multicast tree 的 parent 會傳訊息過來，接受訊息加入 multicast tree，並顯示在 UI 上。

被拒絕：收到拒絕訊息，顯示在 UI 上。

◆ Leave Service provided by Others :



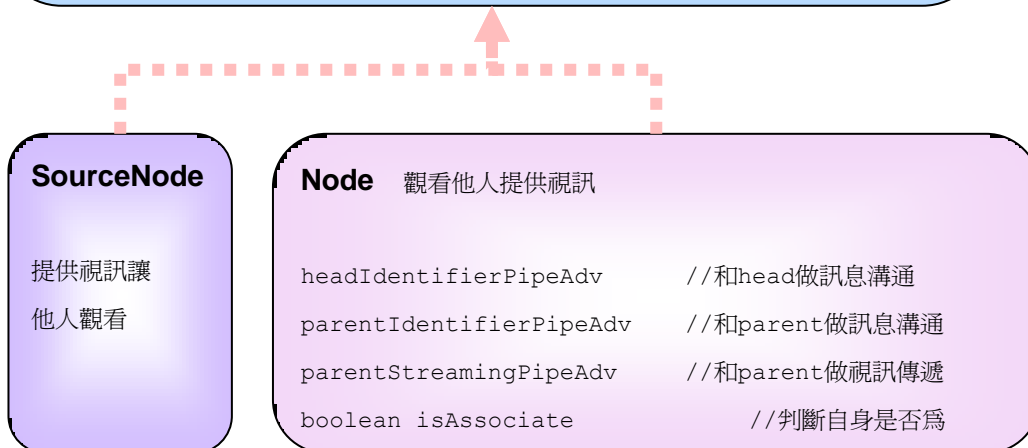
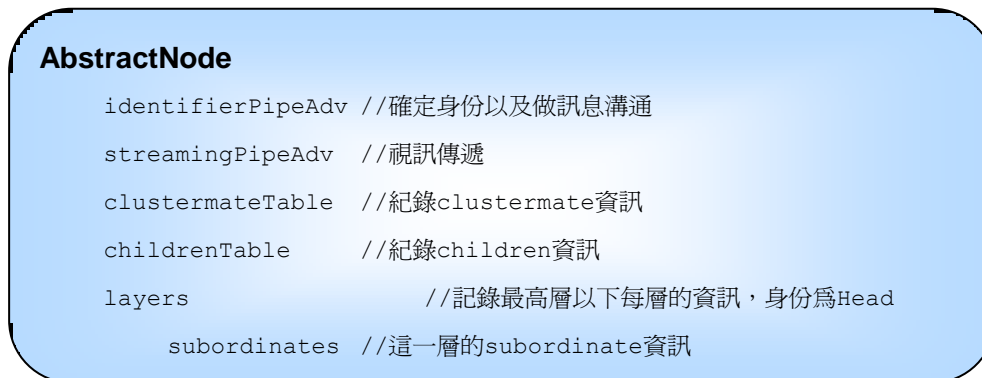
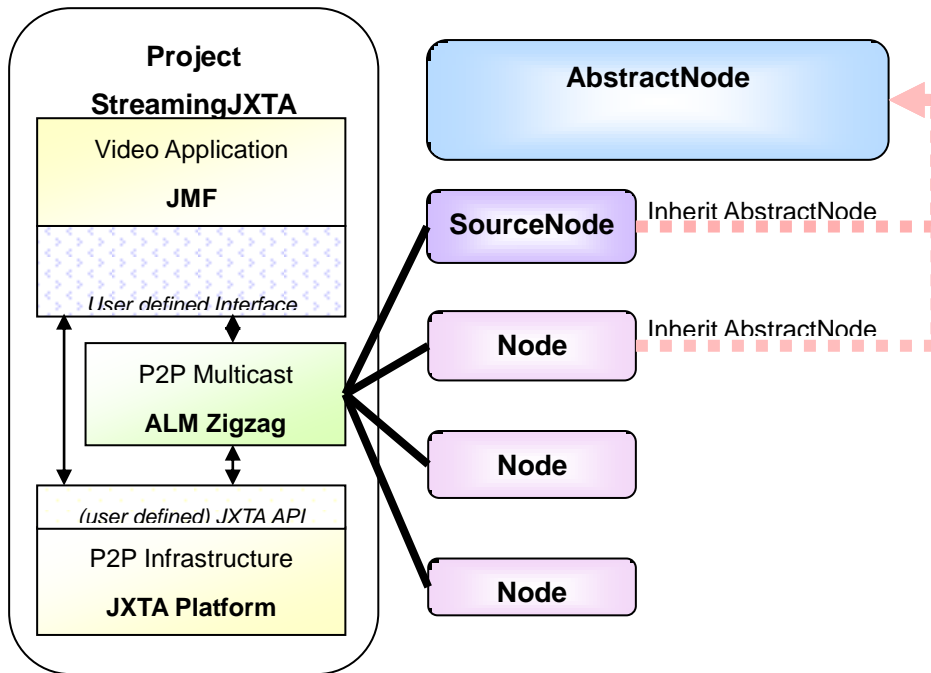
1. 在 UI 上點選離開指定 Service 並傳送離開訊息給所有相關的 tree node，讓他們可以做出處理。

2. 傳送離開訊息給 service provider，顯示不再觀看訊息。

3. 關閉 multicast tree。

### 3. Low level design

- Data structure UML



- Join Algorithm

Join 是當一個新的 peer 要加入這個群播樹狀結構時所需要做的動作，它要做的事情有：找到最適合的末端，並將要加入的 peer 加入至那個相對應的 cluster 中。它的演算法如下，其中 X 為接收到 join request 的 peer，P 為欲加入的 peer：

```
If X is a layer-0 associate head
  Add P to the only cluster of X
  Make P a new child of X
Else
  If Addable(X)
    Select a child Y:
      Addable(Y) and  $D(Y) + d(Y, P)$  is min
    Forward the join request to Y
  Else
    Select a child Y:
      Reachable(Y) and  $D(Y) + d(Y, P)$  is min
    Forward the join request to Y
```

- Depart Algorithm

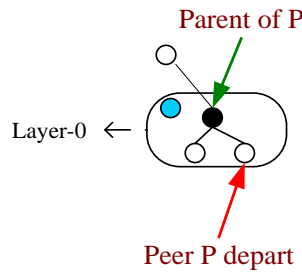
Depart 是當一個 peer 從群播樹狀結構中離開時，為了維持樹狀結構的完整所需要做的動作。簡單來說，需要處理的事情主要有：

1. 它的 parent 把與此離開 peer 的 link 刪掉
2. 如果這個離開的 peer 有 child 是藉由它取得影像的話，則那些 children 就必須找到新的 parent
3. 如果這個離開的 peer 的最高層數不是最底端的話，那就必須找到新的 peer 來取代離開的 peer 在各層中的角色。

而一個 peer 離開樹狀結構有可能是在主動的情況下，像是使用 service 的人決定結束使用這項服務；相對的，peer 的離開也有可能是屬於被動的情況，例如因為網路的不穩定，而被迫中斷使用這項服務，不管是在主動或是被動的狀態下離開，處理上的差別只在於是由哪一個（或一些）peer 來負責做到上述三項事情。以下分別解釋各種情況下的處理方式：

假設離開的 peer P 之最高層為 Layer-i

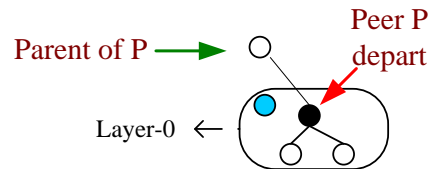
**A.  $i = 0$  , P 為 non-associate head node (如下圖)**



P 為 Layer-0 non-associate head node

在這個情形下，所需要做的改變只有 P 的 parent 要將它與 P 的連結刪除即可。(即只需要做上面所說的事項 1.)

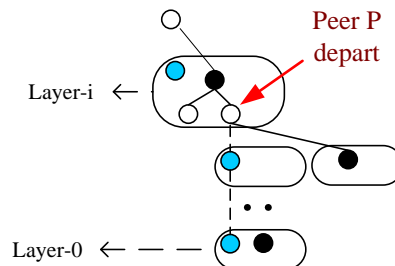
**B.  $i = 0$  , P 為 associate head node (如下圖)**



P 為 Layer-0 associate head node

這時所需要做的改變除了它的 parent 需要將它與 P 的連結刪除之外(事項 1.)，還必須處理 P 與它的 children 之間的連結(即事項 2.)。由於 P 在這個 cluster 中扮演的是 associate head 的角色，當它離開後，就必須從其他 non-head nodes 中，選出一個來取代 P 成為 associate head。在主動的情況下，便是由 P 來選出，而在被動的情況下，便是由 cluster head 來決定，一旦決定以後，原本屬於 P 的 children 便成為新的 associate head 的 children，由此方法來取代 P 與它的 children 之間的連結。

**C.  $i > 0$  , P 為 non-associate head node (如下圖)**

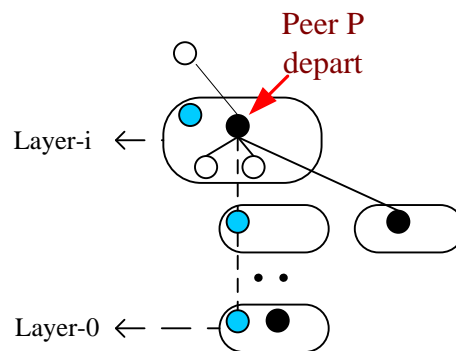


P 為 Layer-i ( $i > 0$ ) non-associate head node

在這個情形下，就必須做到上面所說的三個事項，

1. 就是 parent 刪除與 P 的連結
2. 在處理 P 與它的 children 的連結時，則交由各個 child 所屬的 cluster 的 head 來負責找到新的 parent。也就是說，這些 associate head 所屬的 cluster 的 head 需要在 layer-i 中找尋一個 clustermate 來取代 P 而成為它的 associate head 的新 parent，以維持 ALM 的樹狀結構。
3. 在主動的情況下，則由 P 在它的 layer-0 cluster 中，找出一個 non-associate head 的 node 來取代 P 在 layer-0~layer-i 中的每個位置。如果是被動的情形下，則由它的 layer-0 的 associate head 來負責選出取代 P 的 node。

**D.  $i > 0$ ，P 為 associate head node (如下圖)**



P 為 Layer-i ( $i > 0$ ) associate head node

這個情形也是要做到上面所說的三個事項，

1. 即 parent 刪除與 P 的連結
2. 在處理 P 與它的 children 的連結時，由於它的 children 有兩種，所以處理的方式也不同。第一種 children 為 layer-(i-1) 的 associate head，此種 children 的處理方式與 case C 中的 2. 相同；而另一種 children 則是同屬於 layer-i 的 non head nodes，這種 children 的解決方式與 case B 中相似，也就是等到取代 P 的 node 出現時，就將 parent 改為那個取代 P 的 node，而在新的 parent 還沒出現前，這些 children 便暫時將 cluster head 作為它們的 parent。
3. 這部份的動作便與 case C 中的 3. 相同，也是要找出一個 node 來取代 P 在各層中的位置與角色。

- Split Algorithm

在 ALM Tree 的基本架構中，每個 cluster 不能超過 3k 個 nodes(包括 head 及 associate head)。因此，當一個 cluster 的 size 太大時，就要進行 split，從原有的 cluster 中分出一些 nodes 成為新的 cluster，使兩個 cluster 的 size 都不超過 3k，同時也可以分攤原本 associate head 的負擔。

Split 的過程粗略來分，可以分成三個步驟。在接下來的示意圖中，圓角矩形代表 cluster，灰色圓圈代表該 cluster 的 head，黑色圓圈代表它的 associate head，白色圓圈則代表普通的 node(即非 head 也非 associate head)。Figure 1-1 是 cluster U 進行 split 前的架構，它的 head 是  $X_{head}$ 、associate head 是  $X_{assoc}$ 。

Split 的第一步就是把 cluster U 的一些 nodes 移到新的 cluster V 裡，如 Figure 1-2。

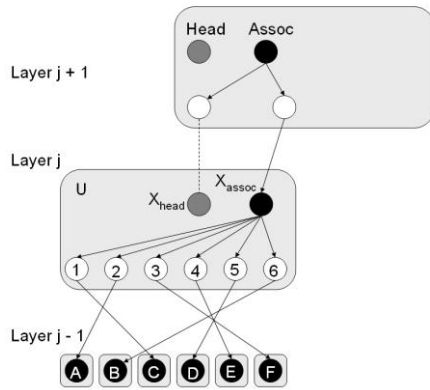


Figure 1-1  
Cluster U split 前

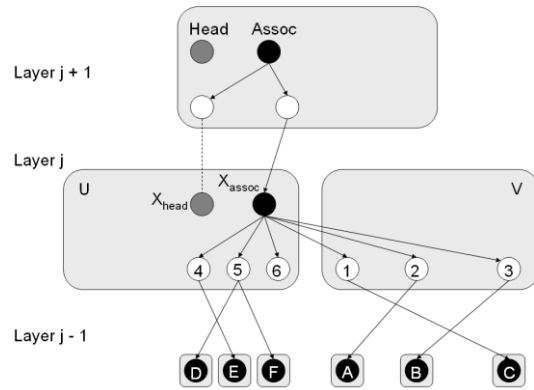


Figure 1-2  
將 cluster U 的 nodes 移至 cluster V

第二步是從 cluster V 的 nodes 裡選出 head 及 associate head，分別稱為  $X'_{head}$  與  $X'_{assoc}$ 。指定完成後就會變成 Figure 1-3 的樣子。

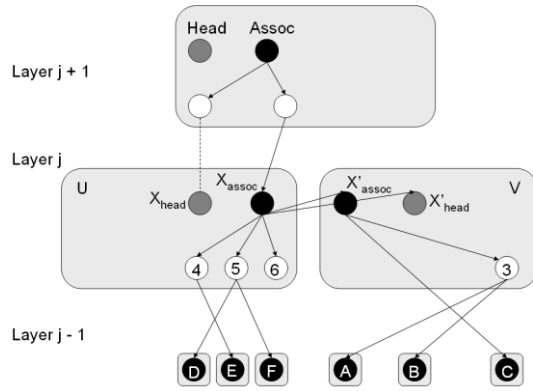


Figure 1-3

指定 cluster V 的 head 及 associate head

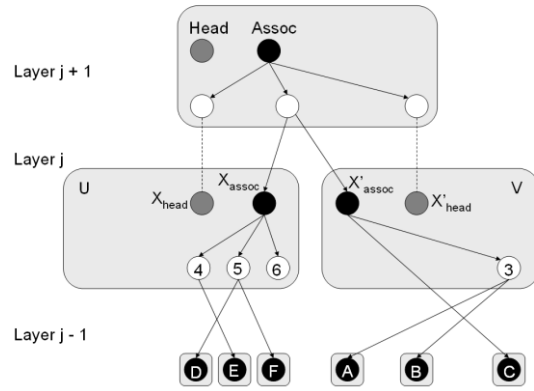


Figure 1-4

將  $X'_{head}$  加到上一層的 cluster 中

最後，把  $X'_{head}$ ，加到上一層的 cluster 裡，當作普通的 node，如 Figure 1-4，這個 split 的動作就算完成了。

至於 split 詳細的過程則會根據 cluster 所在的 layer 而有所差異，主要可以分成三種不同的情況：在 layer 0、在 layer 1 ~ layer H-2 以及在 layer H-1。其中 H 是整個 tree 的高度，而 layer H-1 就代表最高的那層。

為了方便說明，同樣假設進行 split 的 cluster 為 U，它的 head 是  $X_{head}$ 、associate head 是  $X_{assoc}$ ，產生的 cluster 稱做 V，它的 head 及 associate head 分別為  $X'_{head}$  與  $X'_{assoc}$ ，而 S 代表著 Source(同時為 layer H-1 的 head 及 associate head)。

A. 當進行 split 的 cluster 在 layer 0 時，步驟如下：

1. Figure 2-1 裡， $X_{head}$  發現 cluster 太大時，會將它在 layer 0 的 subordinates(不包括  $X_{assoc}$ )依據它們到 Source 的 delay<sup>1</sup>做排序，拿 delay 較大的那一半 subordinates 作為新的 cluster。見 Figure 2-2。

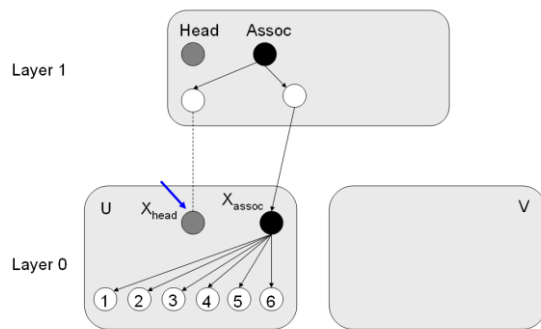


Figure 2-1

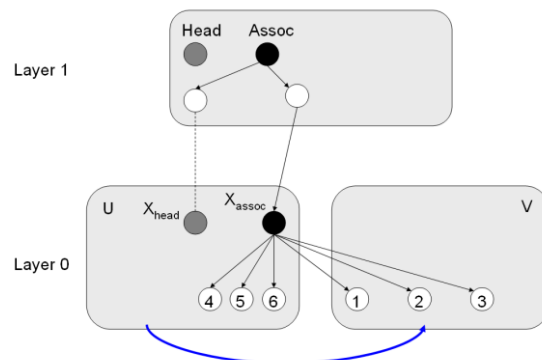


Figure 2-2

<sup>1</sup> Delay 的計算並沒有明確的定義。

2.  $X_{\text{head}}$  會指定 delay 最大的為 cluster V 的 head，delay 第二大的為 cluster V 的 associate head，也就是  $X'_{\text{head}}$  和  $X'_{\text{assoc}}$ 。然後把 cluster V 中其他 clustermates 的 parent 從  $X_{\text{assoc}}$  換成  $X'_{\text{assoc}}$ 。如 Figure 2-3。

3. 接著  $X'_{\text{head}}$  會被加到上一層的 cluster 中，成為其中一個 subordinate，並且它的 parent 也會被改成上一層 cluster 的 associate head。如 Figure 2-4。

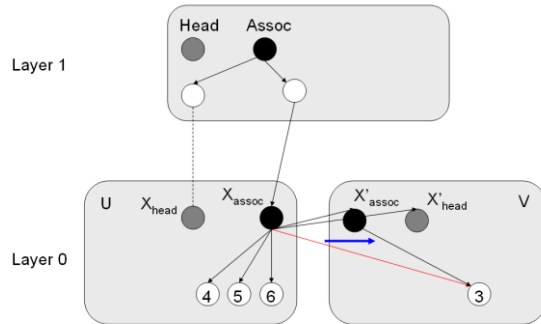


Figure 2-3

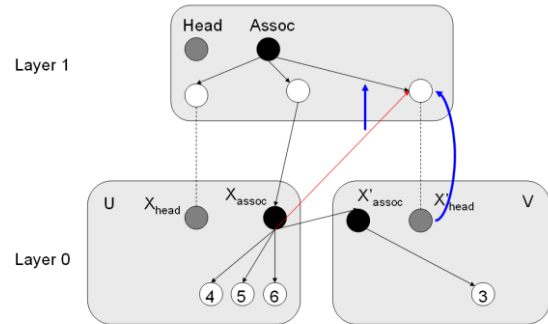


Figure 2-4

4. 最後在 Figure 2-5 中，把  $X'_{\text{assoc}}$  的 parent 改成和  $X_{\text{assoc}}$  的 parent 一樣，split 的過程就完成了。見 Figure 2-6。

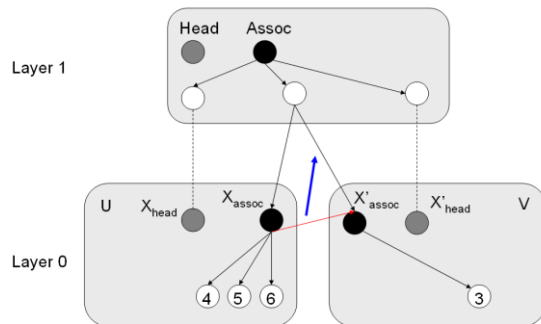


Figure 2-5

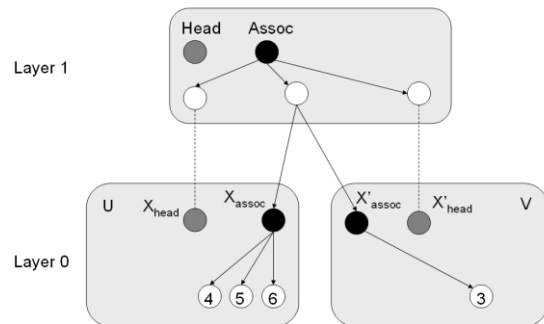


Figure 2-6

B. 當進行 split 的 cluster 在 layer  $j$ ，且  $0 < j < H-1$  時，步驟如下：

1.  $X_{\text{head}}$  發現 cluster 太大時，會將它在 layer  $j$  的 subordinates (不包括  $X_{\text{assoc}}$ ) 拿一半去產生新的 cluster<sup>2</sup>。

<sup>2</sup> 將 subordinates 分成兩半時，應該要盡可能的讓每一個 subordinate 的 children 的 heads 與該 subordinate 在同一部分，以減少重新指定 parent 的動作。但由於沒有明確且有效率的演算法，目前只有隨機地分群。

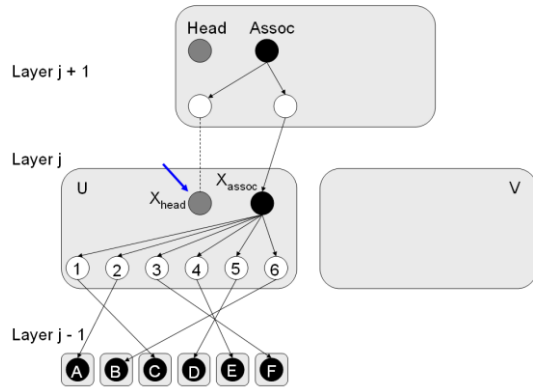


Figure 3-1

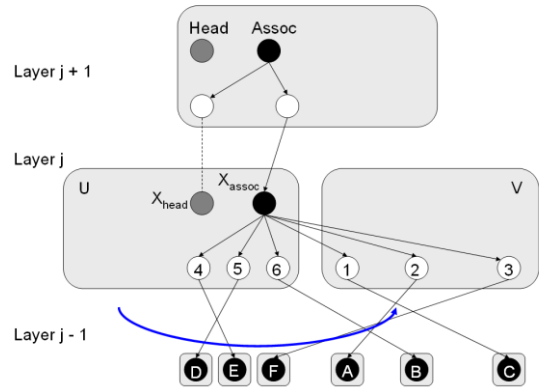


Figure 3-2

2. 分群完畢後，檢查是否有 subordinate 它的 layer j-1 children 的 head 與該 subordinate 在不同的部分。若有 layer j-1 child 的 parent 和 head 在不同部分，便重新指定該 child 的 parent，使它的 parent 與它的 head 屬於同一部分。

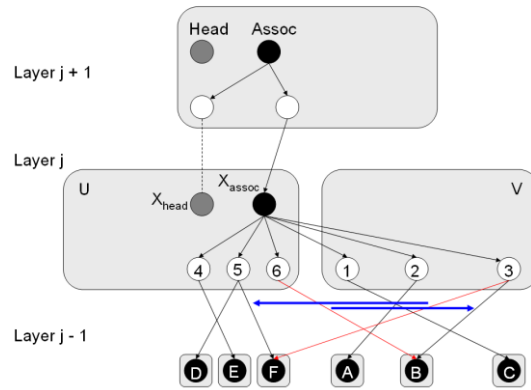


Figure 3-3

3. 接著指定 cluster V 裡 degree 最小的 clustermate 為  $X'_{head}$ ，並重新指定  $X'_{head}$  的所有 children 的 parent，改為 cluster V 裡的其他 node。再從 cluster V 中選 degree 第二小的 clustermate 作為  $X'_{assoc}$ ，把其他在 cluster V 的 clustermates 的 parents 由  $X_{assoc}$  改成  $X'_{assoc}$ 。

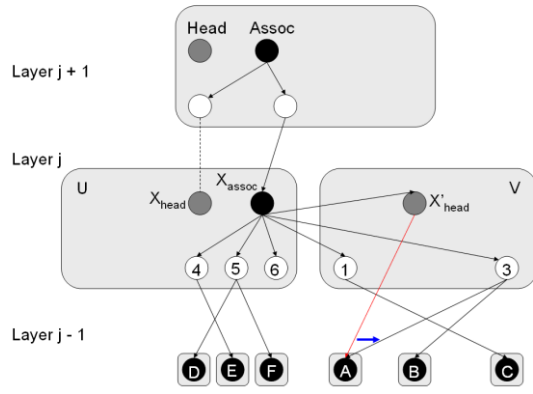


Figure 3-4

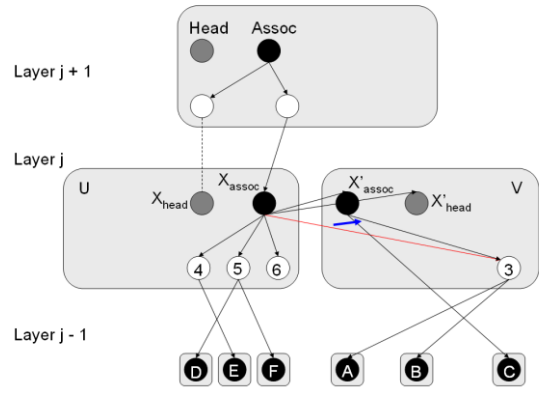


Figure 3-5

4. 剩下的兩個動作就和第一種情況的步驟 3 到步驟 4 一樣。 $X'_{head}$  會被加到上一層的 cluster 中，成為其中一個 subordinate，並且它的 parent 也會被改成上一層 cluster 的 associate head。

5. 最後，將  $X'_{assoc}$  的 parent 改成和  $X_{assoc}$  的 parent 一樣就完成了。

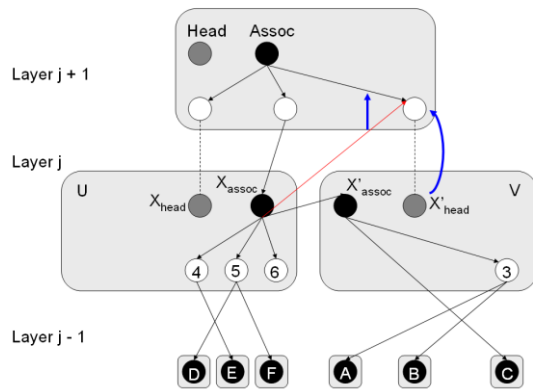


Figure 3-6

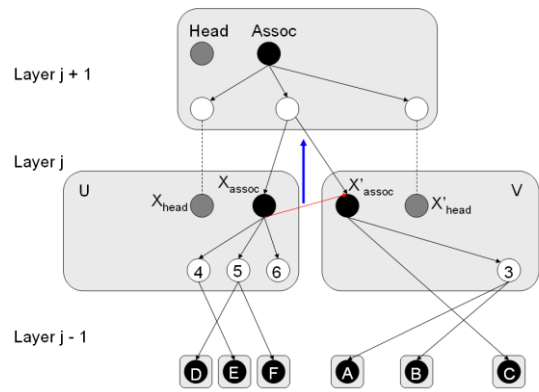


Figure 3-7

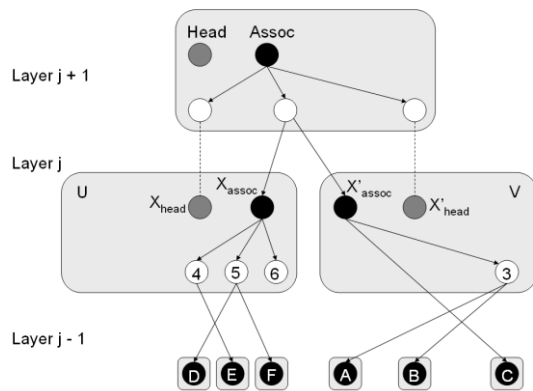


Figure 3-8

C. 當進行 split 的 cluster 在 layer H-1 時，步驟如下：

1. 一開始的三個步驟與第二種情況幾乎相同。S 發現 cluster 太大時，會將它在 layer H-1 的 subordinates 拿一半去產生新的 cluster。

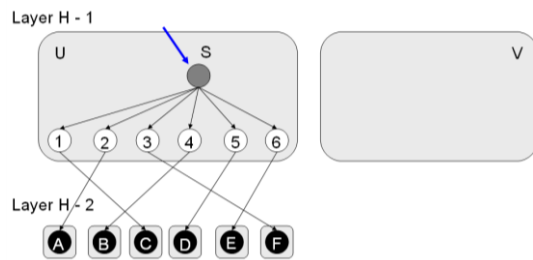


Figure 4-1

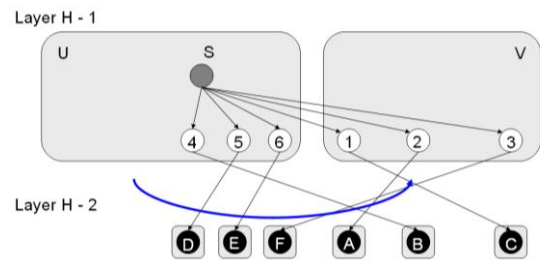


Figure 4-2

2. 分成兩部分後，檢查是否有 subordinate 它的 layer H-2 children 的 head 與該 subordinate 在不同的兩邊。若有這種情形，便重新指定該 child 的 parent，使它的 parent 與它的 head 屬於同一邊。

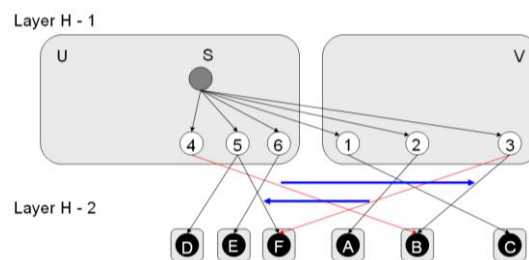


Figure 4-3

3. 接著指定 cluster V 裡 degree 最小的 clustermate 為  $X'_{head}$ ，並重新指定  $X'_{head}$  的所有 children 的 parent，改為 cluster V 裡的其他 node。再從 cluster V 中選 degree 第二小的 clustermate 作為  $X'_{assoc}$ ，把其他在 cluster V 的 clustermates 的 parent 由 S 改成  $X'_{assoc}$ 。

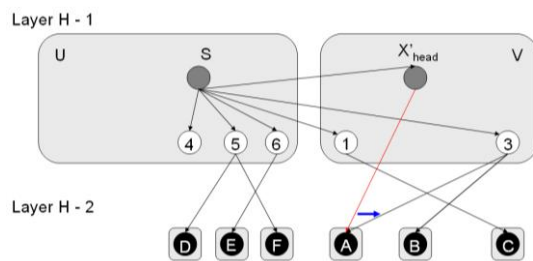


Figure 4-4

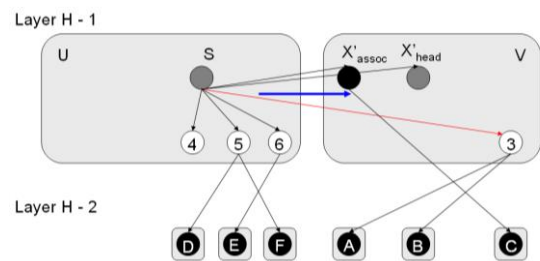


Figure 4-5

4. S 會產生的新的 cluster 在 layer H，並把自己指定為該 cluster 的 head 及 associate head。

5. S 會從 cluster U 的 subordinates 中挑選 degree 最小的作為 cluster U 的新 associate head(即圖中的 node A)，並將它在 cluster U 的所有 children 的 parent 改成 A。

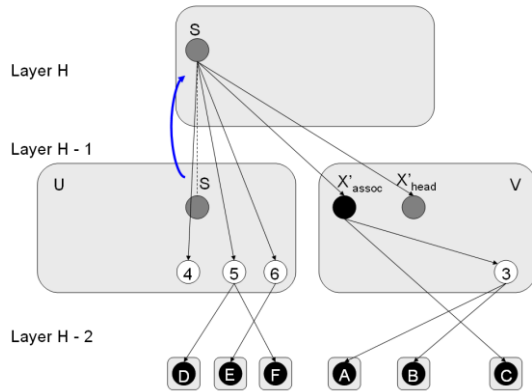


Figure 4-6

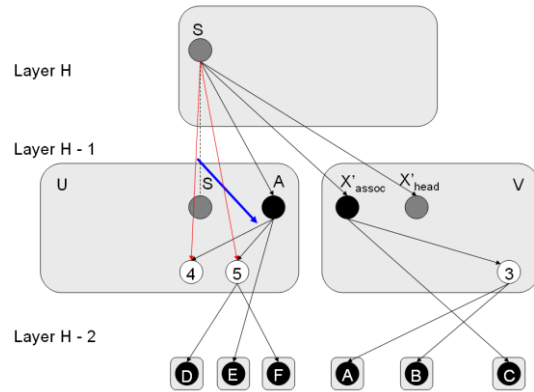


Figure 4-7

6.  $X'_{head}$  會被加到 layer H 的 cluster 中，成為一個 subordinate，並把 A 的 parent 從 S 改成  $X'_{head}$ ，split 就完成了。

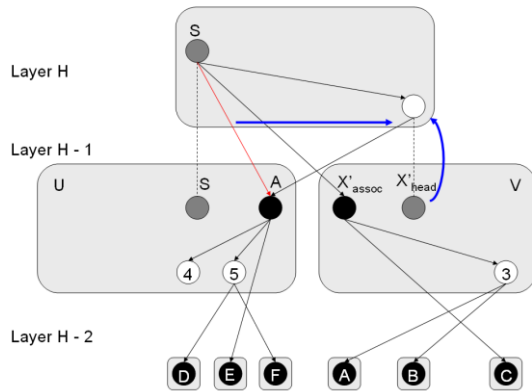


Figure 4-8

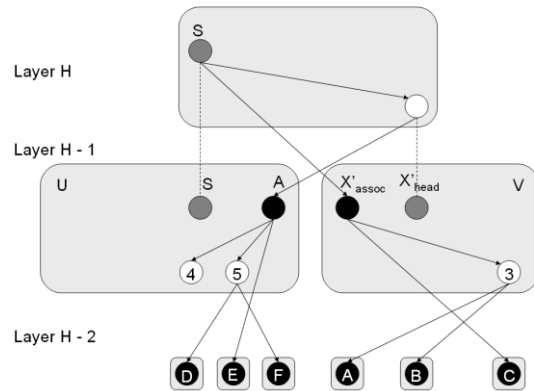


Figure 4-9

- Merge Algorithm

當一個 layer-j cluster U 的 size 小於 k 時，為了維持每個 cluster 的最小 size 以便使 ALM Tree 正常運作(例如：node 越來越少，最終整個 cluster 機能無法運作而 crash)，這時必須與另外一個 cluster V 做合併動作。選取 V 必須遵循以下幾個限制：

- 和 U 擁有相同的 super cluster。
- 是 layer-j 中符合上述條件最小的 cluster(除了 cluster U)

外)。

Figure 1-1 是 cluster U 和 cluster V 進行 Merge 前的架構，head 和 associate head 分別為  $U_{head}$ 、 $U_{assoc}$  和  $V_{head}$ 、 $V_{assoc}$ 。

Merge 的過程大略可分成以下三個步驟：

1. 先從 U 和 V 的 head 和 associate head 中挑取新的 head  $(U+V)_{head}$  和 associate head  $(U+V)_{assoc}$ ，如 Figure 1-2 中新的  $(U+V)_{head}$  和  $(U+V)_{assoc}$  為舊有的  $U_{head}$  和  $V_{assoc}$ 。

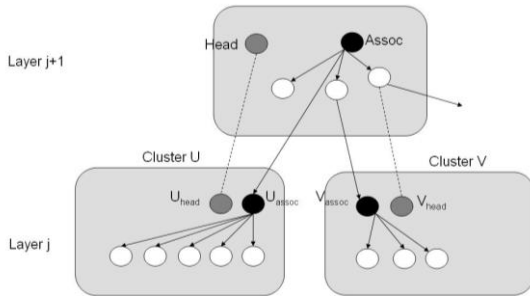


Figure 1-1

Cluster U 和 Cluster V merge 前

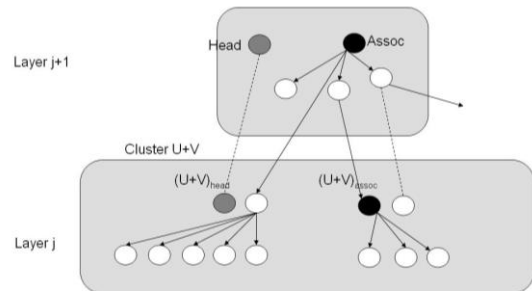


Figure 1-2

新的  $(U+V)_{head}$  和  $(U+V)_{assoc}$ 。

2. 將原先  $U_{assoc}$  的 children(包括自己)重新連到新的  $(U+V)_{assoc}$ ，如 Figure 1-3。
3. 將原先  $V_{head}$  的 children 連到其他 clustermate，並從 layer j+1 拉下，如 Figure 1-4。

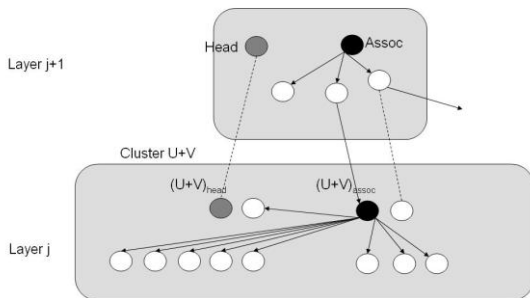


Figure 1-3

Redirect  $U_{assoc}$  Children to  $(U+V)_{assoc}$

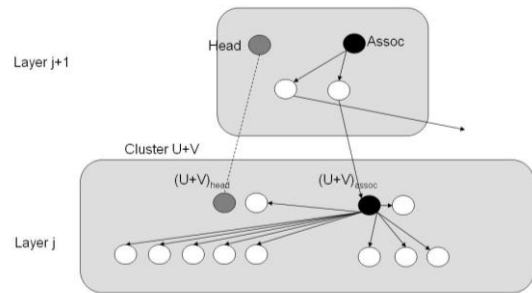


Figure 1-4

拉下  $V_{head}$

Merge 詳細的過程則會根據要求合併的 cluster 和與其合併的另一個 cluster，兩者的 cluster head 在上層的角色來決定。假設現在 layer-j 中 cluster U 要求和 cluster V 合併成 cluster U+V，以下依據 3 種情形做 merge 的詳細說明。

- A. layer-j cluster U 或 cluster V 的 head 是上層(layer j+1)的 head :  
 以下假設  $U_{head}$  也是 layer j+1 的 head。(若是上層的 head 同理)
1.  $U_{head}$  和  $V_{assoc}$  為新的  $(U+V)_{head}$  和  $(U+V)_{assoc}$  , 如 Figure 2-2。

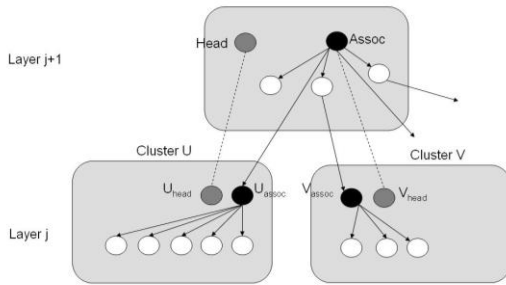


Figure 2-1

Cluster U 和 Cluster V merge 前

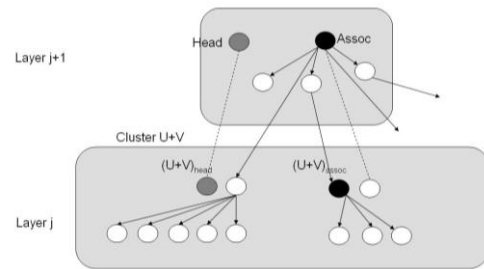


Figure 2-2

新的  $(U+V)_{head}$  和  $(U+V)_{assoc}$ 。

2. 將原先  $U_{assoc}$  的 children(包括自己)重新連到新的  $(U+V)_{assoc}$  , 如 Figure 2-3。
3. 將原先  $V_{head}$  的 children 連到其他 clustermate。若為上層的 associate head, 則也要找一個新的 associate head 並連過去。將  $V_{head}$  從 layer j+1 拉下, 如 Figure 2-4。

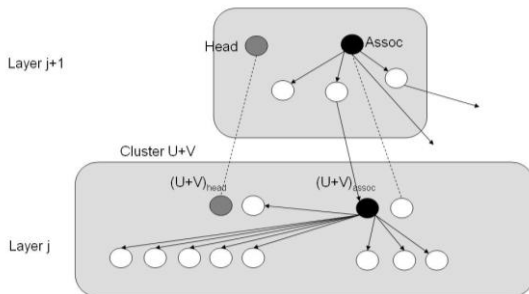


Figure 2-3

Redirect  $U_{assoc}$  Children to  $(U+V)_{assoc}$

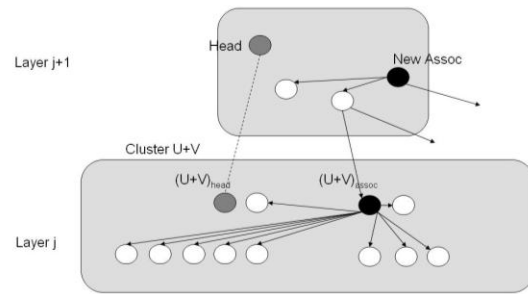


Figure 2-4

拉下  $V_{head}$

- B. layer-j cluster U 或 cluster V 的 head 是上層(layer j+1)的 associate head, 且皆非上層的 head :

以下假設  $U_{head}$  也是 layer j+1 的 associate head。(  $V_{head}$  若是上層的 associate head 同理)

1. 選擇  $U_{head}$  為新的  $(U+V)_{head}$ 。若 cluster U 的 size 大於 cluster V, 則選擇  $U_{assoc}$  為新的  $(U+V)_{assoc}$ , 反之亦然。這裡假設 cluster V 的 size 比較大, 所以選擇  $V_{assoc}$  為新的  $(U+V)_{assoc}$ , 如 Figure 3-2。

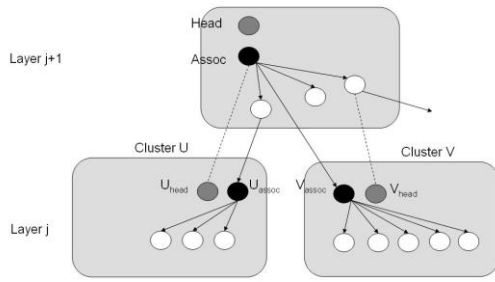


Figure 3-1

Cluster U 和 Cluster V merge 前

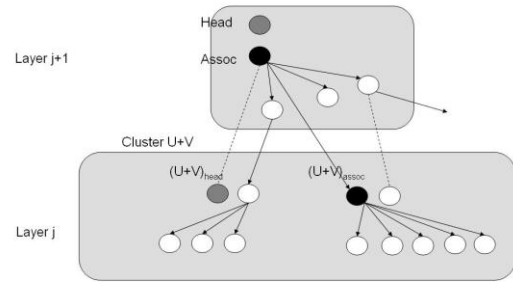


Figure 3-2

新的  $(U+V)_{\text{head}}$  和  $(U+V)_{\text{assoc}}$ 。

2. 將原先  $U_{\text{assoc}}$  的 children(包括自己)重新連到新的  $(U+V)_{\text{assoc}}$ ，如 Figure 3-3。
3. 將原先  $V_{\text{head}}$  的 children 連到其他 clustermate。將  $V_{\text{head}}$  從 layer j+1 拉下，如 Figure 3-4。
4. 若原先  $V_{\text{assoc}}$  的 parent 為  $U_{\text{head}}$ ，則有必要重新為  $V_{\text{assoc}}$  選擇一個新 parent，因為現在  $(U+V)_{\text{head}}$  和  $(U+V)_{\text{assoc}}$  在同一個 cluster 中。只要選擇 layer j+1 中 degree 最少的 nonhead 做 parent 即可，如 Figure 3-5。

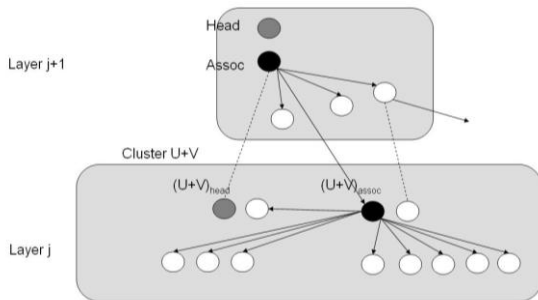


Figure 3-3

Redirect  $U_{\text{assoc}}$  Children to  $(U+V)_{\text{assoc}}$

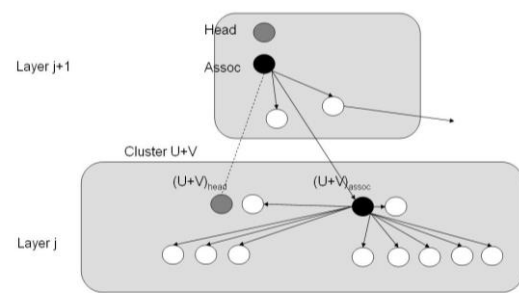


Figure 3-4

拉下  $V_{\text{head}}$

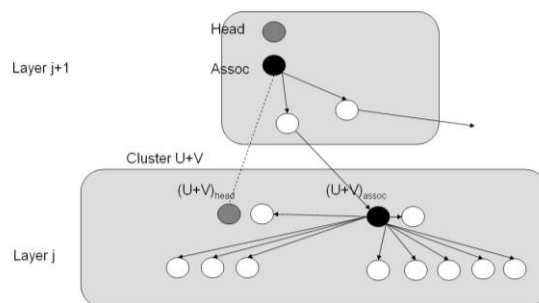


Figure 3-5  $(U+V)_{\text{assoc}}$  連到新 parent

C. layer-j cluster U 或 cluster V 的 head 是上層(layer j+1)的普通 node(none-head 和 none-associate-head) :

1. 若  $U_{head}$  的 degree 高於  $V_{head}$ ，則選擇  $U_{head}$  為新的  $(U+V)_{head}$ ，反之同理。這裡假設  $U_{head}$  的 degree 較高，所以選擇  $U_{head}$  為新的  $(U+V)_{head}$ 。若 cluster U 的 size 大於 cluster V，則選擇  $U_{assoc}$  為新的  $(U+V)_{assoc}$ ，反之亦然。這裡假設 cluster V 的 size 比較大，所以選擇  $V_{assoc}$  為新的  $(U+V)_{assoc}$ ，如 Figure 4-2。

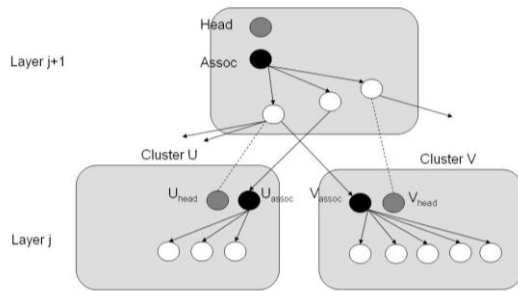


Figure 4-1

Cluster U 和 Cluster V merge 前

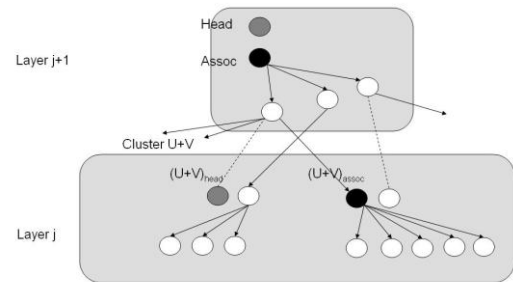


Figure 4-2

新的  $(U+V)_{head}$  和  $(U+V)_{assoc}$ 。

2. 將原先  $U_{assoc}$  的 children(包括自己)重新連到新的  $(U+V)_{assoc}$ ，如 Figure 4-3。
3. 將原先  $V_{head}$  的 children 連到其他 clustermate。將  $V_{head}$  從 layer j+1 拉下，如 Figure 4-4。
4. 若原先  $V_{assoc}$  的 parent 為  $U_{head}$ ，則有必要重新為  $V_{assoc}$  選擇一個新 parent，因為現在  $(U+V)_{head}$  和  $(U+V)_{assoc}$  在同一個 cluster 中。只要選擇 layer j+1 中 degree 最少的 nonhead 做 parent 即可，如 Figure 4-5。

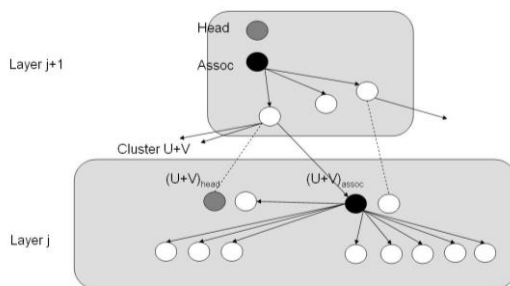


Figure 4-3

Redirect  $U_{assoc}$  Children to  $(U+V)_{assoc}$

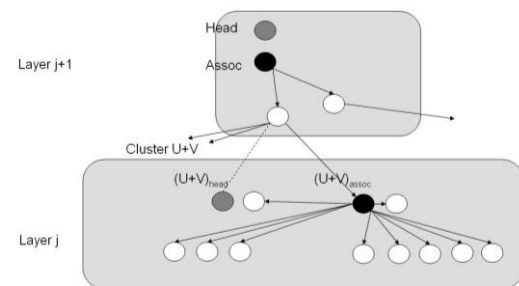


Figure 4-4

拉下  $V_{head}$

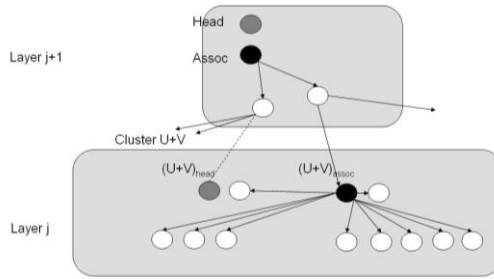


Figure 4-5  $(U+V)_{\text{assoc}}$  連到新 parent

### III. Media Framework

#### ✓ JMF

Java Media Framework API (JMF)是可以一個把語音、影像結合在 Java 程式上的介面。我們利用這個 library 所提供的 classes，擷取 web camera 拍攝到的即時畫面，並將那些畫面轉成一張張的圖片，最後藉由 JXTA 的 pipe 進行發送或接收。

### IV. Communication Over JXTA

#### ✓ EfficientBiDiPipe

JXTA 有提供一種讓 peer 與 peer 間可以雙向溝通的管道，稱為 JxtaBiDiPipe。其中 BiDi 是指 bi-directional，它就是利用 InputPipe 及 OutputPipe 使得 peer 可以接收和傳送訊息。一個 peer 要與別的 peer 建立連線之前，要連的 peer 必須先跑起 JxtaPipeServer，它會去聽要求連線的 request，收到 request 後，它會利用 request 裡的資訊產生一個 JxtaBiDiPipe 連向對方，並傳送 respond 過去。而要求連線的 peer 收到 respond 後，就可以產生 JxtaBiDiPipe 指向另一個 peer，連線的動作就完成了。

然而，在我們的實作上，常常會需要在 JxtaBiDiPipe 建立完成後，就立刻傳送訊息過去，而被連的 peer 也要能在連線一建立就知道對方的身分，才可以進行明確的動作。而原有的 JxtaBiDiPipe 無法達到我們的需求，因此我們把 JxtaBiDiPipe 改寫成我們需要的樣子，稱為 EfficientBiDiPipe。

其實，我們的需求只是在連線建立的同時，也可以傳送其他的訊息。JxtaBiDiPipe 在要求連線時會發出一個 request，而 JxtaServerPipe 也會回應一個 respond。我們便利用 request 及 respond，把要發送的訊息包在裡面，接下來就照著原本 JxtaBiDiPipe 與 JxtaServerPipe 溝通的方式去傳送。最後，再把包在 request 及 respond 裡的訊息給拿出來，就大功告成了。

我們撰寫了 EfficientBiDiPipe 及 EfficientServerPipe，分別是對應 JxtaBiDiPipe 及 JxtaServerPipe。

## V. UI Design

右圖為預先設計的 UI 介面：

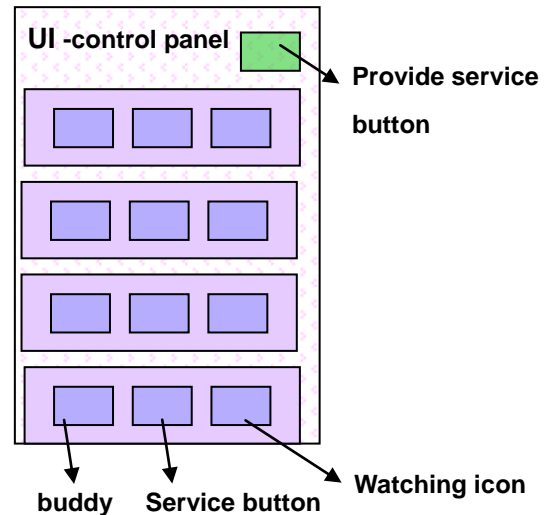
- ◆ 所有淺紫色的框框代表一個加入此 Mutlicast Tree 的 Buddy 以及其資訊。

第一個藍色框代表 Buddy 的名字。

第二個藍色框代表 Buddy 是否提供 Service 讓其他人觀看，壓下可以觀看此 Buddy 提供的 Service。

第三個藍色框代表 Buddy 是否正在看 user 本身提供的 Service，按掉的話可以拒絕此 Buddy 觀看 Service。

- ◆ 綠色按鈕一壓下就可以提供自身 Service (Webcam 視訊) 讓其他 Buddy 觀看。



## 四、計畫成果

### I. Customized JxtaBiDiPipe

#### i. EfficientBiDiPipe

請參閱研究方法，四、Communication Over JXTA。

### II. Our Implementation

#### i. Simplified Join Algorithm

可分為兩種情況，新的 Peer 加入時，會先傳一個 message 給 Server，呼叫 Server 執行 takeJoinRequest。若 Server 所容納的 children 個數還沒到達 MAX\_CHILDREN 時，則 Server 就把新的 Peer 設成自己的新的 child，並且傳回一個 message 給它，叫新的 Peer 去更新 parent 成 Server(見 Fig. a)。

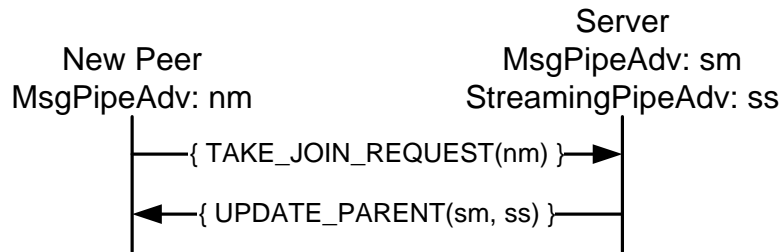


Fig. a 當 Server 還能容納新的 Peer 時

若 Server 的 children 個數已達 MAX\_CHILDREN 時，則 Server 會將收到的 message 傳給其中一個 child 來處理，而該 child 再去看它自己的 children 個數是否已滿。若還沒滿，則把新的 Peer 設成自己的 child，並叫新的 Peer 去更新 parent 資訊(見 Fig. b)。若滿了就繼續傳給它的 child 處理，直到有 Peer 能把新的 Peer 加到自己的 children 中為止。

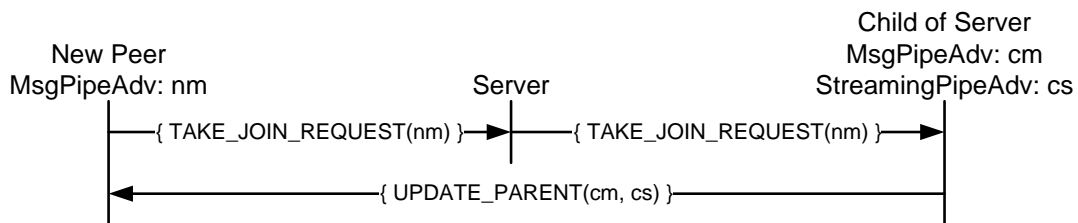


Fig. b 當 Server 的 children 個數已滿時

## ii. Remote Procedure Call / Message Format

### 1. Goal

P2P network 主要是由 Node 之間的溝通來維持，再加上為了維持 ALM tree 的架構，當遇到各種 event 時，Node 除了自己要處理外，常常也需要其他 Node 的參與，所以需要設計 remote procedure call 來通知其他 Node 做出相對的處理。

### 2. Call

#### 甲、Message

我們設計了 Message 以 XML 格式來傳送 RPC request，其格式如下：

```

<Method>
  <Identity>
    <MsgPipeID>
      RPC Request 的來源 Pipe ID，讓接收者得以判別
      Request 來源。
    </MsgPipeID>
  </Identity>
  <Method"X"> (X 代表第 X 個 Method)
  
```

```

    <MethodName>
        所呼叫的 Function Name
    </MethodName>
    <Argument>
        該 Function 所需要的參數
    </Argument>
    ...
    <Argument>
    </Argument>
    </Method"X">
</Method>

```

#### 乙、ValueElement

為了解決 Remote Procedure Call 的參數傳遞問題，我們設計了 ValueElement Class 來包裝/反包裝 ValueType 和 Object。同時將 Vector 轉成 String (或將 String 轉回成 Vector)，以方便在 pipe 上傳輸。

#### 丙、使用方法

```

MessageBuilder mb = new MessageBuilder();
mb.setPipeID(PipeID);
mb.addMethod(funcName, new ValueElement[] { (object,
Type) });

```

再透過 EfficientBiDiPipe 傳送給對方即可。

## 五、 結語與未來方向

### I. 結語

我們目前做出一個 multicast tree 的雛形，可順利在無線網路上運作。每個 peer 可以接收多人的 service，亦可提供自己的影像服務。雖然受限於 Distributed State Consistency 的問題無法有效解決，我們並沒有完成整個 algorithm 的實做。但是我們已經成功的將 JMF 和 JXTA 結合起來，並修改 JXTA 的底層設計，使其符合我們的需求。並且我們已經設計好完整的 Data Structure 和 pseudo-code，只要 Distributed State Consistency 的問題一解決，我們就可以完成整體的實作。

### II. Future Work

#### i. Distributed State Management

目前我們在實做演算法的時候遇到 Distributed State 無法有效管理的問題。若要維持一個 cluster，每個 node 都需要記住其他 node 目前的狀態。目前我們無法 keep 每一個 node 所記錄資訊的

consistency，明年學弟妹的專題將會就這一部份繼續研究。

ii. Video Quality Adjustment

可提供使用者手動調節影像品質或由程式依據依據頻寬限制自動調整。

iii. Mobility Service

目前我們尚未把 mobility 納入設計之中，在 mobility 的環境下，如何讓使用者觀看的影像不會因為移動而中斷，保持影像的流暢度，也是未來可研究的課題。

## 六、 參考資料

**[1] A Peer-to-Peer Architecture for Media Streaming**

Duc A. Tran, Kien A. Hua, Tai T. Do

*IEEE Journal on Selected Areas in Communications, VOL.22, NO.1, 2004*

**[2] JXTA™ Official Website**

<http://www.jxta.org>

**[3] Java Media Framework API (JMF)**

<http://java.sun.com/products/java-media/jmf/index.jsp>

**[4] JXTA v2.3.x: Java Programmer's Guide**

Sun Microsystems, Inc.

<http://www.jxta.org> Apr. 7, 2005.

**[5] Tapestry: A Resilient Global-scale Overlay for Service Deployment**

Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiawicz

*IEEE Journal on Selected Areas in Communications, January 2004, Vol. 22, No. 1.*

**[6] Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications**

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan

*SIGCOMM'01, August 27-31, 2001,*

**[7] Distributed Object Location in a Dynamic Network**

Kirsten Hildrum, John Kubiawicz, Satish Rao and Ben Y. Zhao

*Theory of Computing Systems, March 2004, No. 37, Pgs. 405-440, Springer-Verlag*

**[8] Rapid Mobility via Type Indirection**

Ben Y. Zhao, Ling Huang, Anthony D. Joseph and John D. Kubiawicz

Appears at *Third International Workshop on Peer-to-Peer Systems (IPTPS)*  
San Diego, CA. February 2004.

**[9] Exploiting Routing Redundancy via Structured Peer-to-Peer Overlays**

Ben Y. Zhao, Ling Huang, Jeremy Stribling, Anthony D. Joseph and John D. Kubiatoiwicz

Appears in *Proceedings of 11th IEEE International Conference on Network Protocols (ICNP)*

**[10] Tapestry: A Resilient Global-scale Overlay for Service Deployment**

Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatoiwicz

*IEEE Journal on Selected Areas in Communications, January 2004, Vol. 22, No. 1.*

**[11] OceanStore: An Architecture for Global-scale Persistent Storage**

John Kubiatoiwicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gumadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells and Ben Zhao.

*Proceedings of ACM ASPLOS, November, 2000*

**[12] A Survey of Peer-to-Peer Content Distribution Technologies**

Stephanos, Routsellis-Theotokis, Diomidis Spinellis

*ACM Computing Surveys, Vol. 36, No. 4, December 2004.*

**[13] A Survey and Comparison of Peer-to-Peer Overlay Network Schemes**

Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steven Lim

*IEEE Communications Survey and Tutorial, March 2004.*

**[14] A Scalable Content-Addressable Network**

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp

*SIGCOMM'01, August 27-31, 2001, San*

## 附錄一、

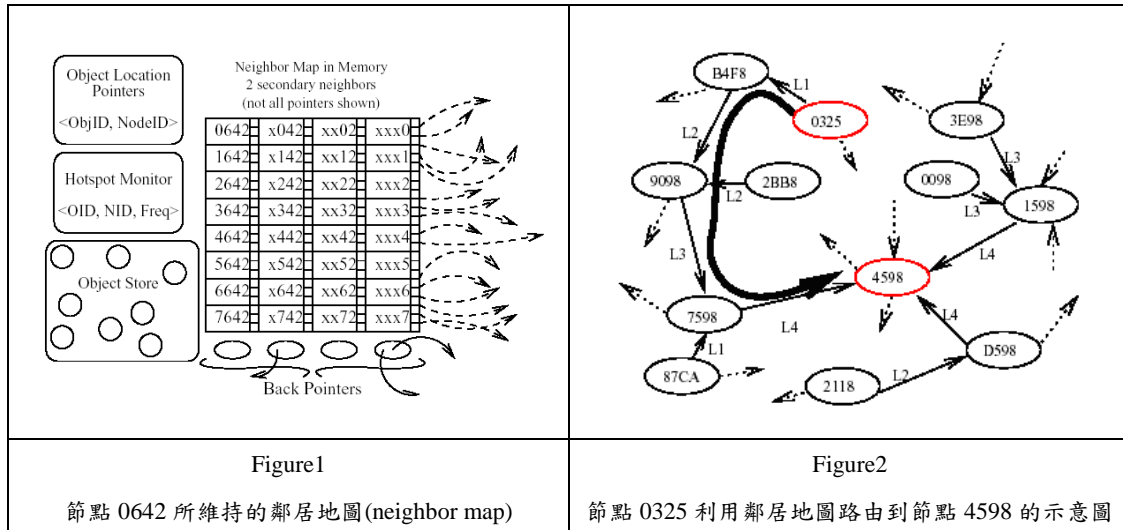
### ➤ Tapestry 和 Chord

#### 1. Tapestry

Tapestry 是一個由加州柏克萊發展的同儕網路，在 Tapestry 的架構之下，整個網路的拓撲就像是一顆樹(tree)。利用節點識別碼 (node ID)，Tapestry 可以保證在  $O(\log n)$  的時間複雜度之內找到所要路由的目標節點。Tapestry 架構之中幾個重要的結構與行為如下：

- 鄰居地圖(neighbor maps)

每一個節點之中會維持一個鄰居地圖(如 Figure1)。其目的是要讓找尋節點的每一個步驟可以在  $O(1)$  之中完成(e.g.:  $***8 \rightarrow **98 \rightarrow *598 \rightarrow 4598$ )。



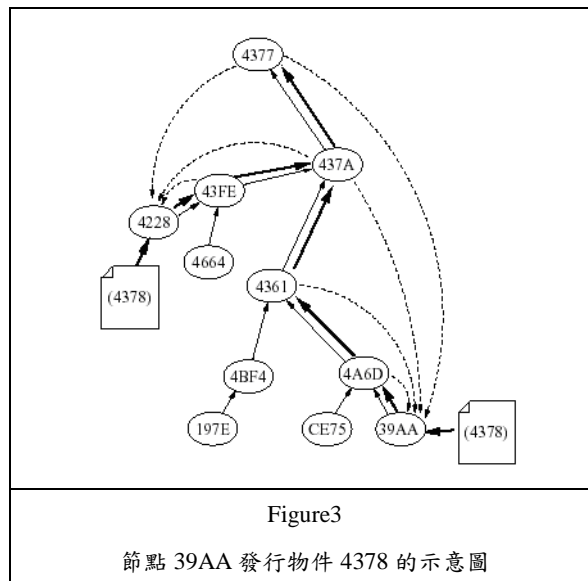
- 尋找節點

Tapestry 利用存在節點之內的鄰居地圖(neighbor maps)，digit by digit 的快速對應找到目標節點。如 Figure2 中，節點 0325 要路由到節點 4598 時，會先找到  $***8 \rightarrow$  再找到  $**98 \rightarrow$  再找到  $*598 \rightarrow 4598$  (‘\*’ 表示可對應到任何數字)，如此就完成了路由程序。

- 物件(object)的發行(publish)

在 Tapestry 中，節點識別碼(node ID)和物件識別碼(object ID)是分開的兩個集合。當一個物件加入這個架構時，物件所在的節點會先利用 MapRoot(Object)這個函數找出負責這個物件的根節點(RootNode)，並負責將這個物件的存在公布給該根節點知道。公布的過程中，所經過的每一個節點也都會有一個

物件指標指向該物件(如 Figure3)，物件所在的節點找到負責物件的根節點之後，根節點會將一個物件指標指向該物件。如此便完成了物件公布的程序。



- 尋找物件

當一個節點要找尋某個 Object 時，會先利用 MapRoot(Object)來找出負責該 Object 的根節點，再利用上述所提到的『尋找節點』的方法路由到該節點，便可經由該節點的物件指標找到該物件。

## 2. Chord

Chord 是由 MIT 所開發出來，能夠提供大規模同儕搜尋服務的底層架構。Chord 利用雜湊函數給每一個節點和每項資料一個獨特的識別碼，結構上會形成一個圓形，如 Figure4，綠點代表目前的節點，而矩形代表著各項資料。圓上的每一個識別碼都會有負責的節點，這些節點會記錄各自該負責的識別碼所對應到的資料的位置資訊，因此使用者在尋找某項資料時，會去詢問負責該資料識別碼的節點，該節點就會回傳那項資料的位置資訊，讓使用者能到存放該資料的地方存取資料。為了能正確地找到應該詢問的節點，圓上的每一個節點都會記錄一個表(finger table)，如 Figure5 所示，finger table 會記錄圓上的哪一段是哪個節點所負責的，並且利用特殊的分段方式，能夠非常有效率地找到所要找的節點。

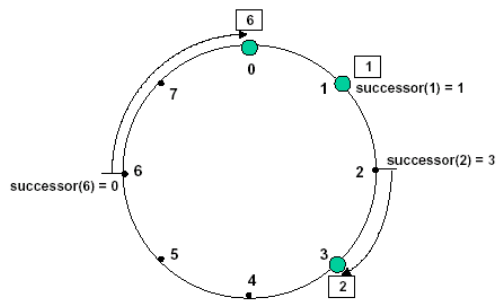


Figure4  
Chord 架構示意圖

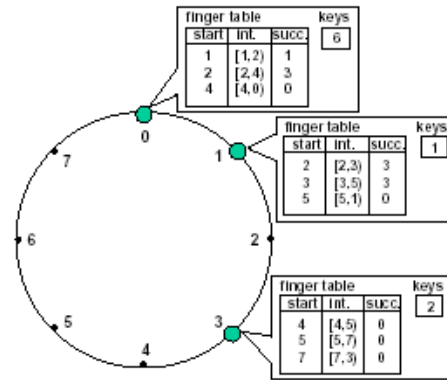


Figure5  
finger table 示意圖

Chord雖然能夠提供很有效率的同儕搜尋服務，但在無線網路上，各個節點的識別碼會因它位置的移動而改變，使得整個結構會有更動，Chord就要花許多時間等待其他的節點更新它們的finger table，效能就會變差。除了不適用在無線網路外，Chord因為每個節點所記錄的資訊很少，大多需要參考其他節點的資訊，就會有安全性上嚴重的問題。有惡意的節點可以很容易地影響其他節點的finger table，甚至使其他節點一定會把資訊傳送給它，從中竊取重要的訊息。這些問題就是我們不選用Chord來當作底層架構的原因之一。