第三章

多人虛擬環境中的角色互動與對話管理

在 2.5.2 我們曾提到 VoiceXML 的對話管理模型在許多研究被廣為採用，許多研究針對 VoiceXML 對話模型不足部份加以增強。但是這些延展 VoiceXML 對話模型的相關研究所提出的對話模型都不適用於多人環境，最大的原因是 VoiceXML 規格設計時僅考量在電話系統中使用，故其對話模型本身就是一個單人環境的對話模型，對話角色並不會改變。

在原始的 VoiceXML 對話模型中，對話參與者限定只有二方，一個是「人」、一個是「系統」，因此在多人環境下此模型必須加以擴充與修正。在本章中，我們會針對此一限制加以分析探討，針對 VoiceXML 加以擴充，提出一個可在多人環境下運作的對話模型。

3.1 動畫與語音訊息在虛擬環境中之傳播

在這一節中我們將討論 Client 如何將自己的動作(如走動、揮手)和語音以腳本方式傳播給其它 Clients。一開始會先考慮較簡單，沒有對話(Dialog)互動的情況。比方說「A 邊揮手邊說 Hello，其它人也可以看得到、聽得到 A 邊揮手邊說 Hello」；而「A 向 B 揮手並說 Hello，B 也向 A 說 Hello，其它人看到、聽到他們二個人在講話」這種情況將在 3.2 節才會討論。

第一種狀況中，Client 會從自身發出一段腳本(Script)，以一對多方式傳送至其它 Clients 播放。我們稱這種只單純廣播，在各 Client 直接放出來，且沒有互

動的 Script 為 *Broadcasting Script*；反之，會等待並取得使用者回應的 Script 稱為 *Dialog Script*。

最後我們將藉此引出主體(Subject)與觀察者(Observer)的概念，這是處理對話時的重要依據。簡言之參與對話的人(主體)看到的 Script(Dialog Script)和其它人(觀察者)看到的 Script(Broadcasting Script)是不一樣的。這個結論將是有關對話互動討論的重要基礎。

我們在探討過程中，將以符號的方式來做對話模型的討論。如表 3.1，假設在虛擬環境中由人類操作的 Client 為 U(User)，則在虛擬環境中由人類所操作的 Client 實例(instance)可以 Ui 來表示，正在進行對話的 Client 以 Us(s 代表 Subject，指正在被討論的主體)表示。系統操作的 Client 為 S(System) ，則在虛擬環境中由系統所操作的 Client 實例可以 Si 來表示，正在進行對話的 Client 以 Ss 表示。

表 3.1　多人環境對話模型的符號及其意義

符號	意義
U	在虛擬環境中由人類所操作的 Client 實例(instance)
S	在虛擬環境中由電腦所操作的 Client 實例
下標 s	主體(Subject)，指正在參與對話的 Client 實例
下標 i	旁觀者(Observer)，指旁觀這段對話的 Client 實例
Us	由人類所操作的 Subject
Ss	由電腦所操作的 Subject
Ui	由人類所操作的 Observer
Si	由電腦所操作的 Observer

3.1.1 Client 與 Client 間腳本(Scripts)的傳播

在此首先考慮的是人類操作的 Client(U)如何將動作、語音傳播給其它 Clients。若對話由系統操作的 Client(S)主動時，也是以類似方式傳播。

當一個參與對話的人類扮演的Client(Us)揮手之後，Us 會將自己揮手的Script 傳給 Server(如圖 3.1，步驟一)，而 Server 幫他轉送給其它線上旁觀的 Clients 播放(如圖 3.1，步驟二)。這種方式其實是大部份 Client-Server 式的多人虛擬環境的運作方式。這種情況下 Script 是由主體(Subject)送給其它旁觀者(Observer)。在本例中，主體是 Us，旁觀者是 U_1 與 U_2。

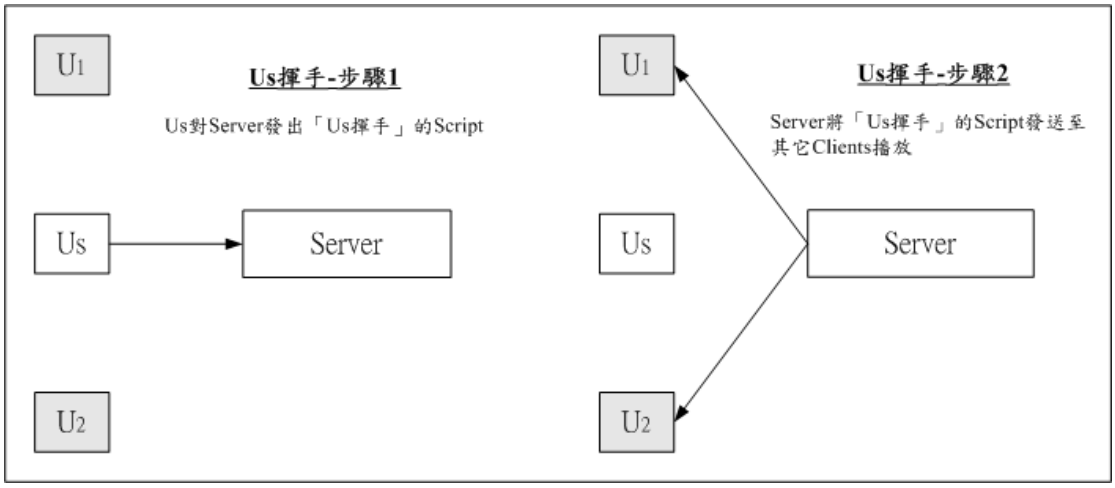

圖 3.1　由人類扮演的 Client 如何將動作(動畫)以 Script 的方式傳送給他其它 Clients。

其中 Us 為主體(Subject)，U1 和 U2 為旁觀者(Observers)。

值得注意的是，主體 Us 發出的訊息目標不需要包含自己，也就是說本身不需要接收自己發出的這段 Script。這種實作方式是基於二個考量。首先是這種實作方式不需檢查那些訊息要忽略，那些不用，可以讓 Client 的設計簡化，其次是可以降低不必要的訊息量。但在多人環境下使用此種做法會使對話無法正常運

作，因為參與對話的二個人之間所傳送的是這二個人對話用的 Script，不應該被其它旁觀者看到。因此「參與對話」的人和「旁觀者」所看到的 Script 必須加以區分。

3.1.2 語音腳本的傳播

由人類扮演的 Client 將動作及語音以腳本(Script)的方式傳送給其它 Client 的方式和沒有語音時的情況做法其實幾乎一模一樣。不同的是此時所有的 Client 必須具有語音辨識(ASR)與語音合成(TTS)的機制。

主要運作方式為 ASR 能夠將使用者的自然語音轉成文字，由 Client 程式包裝成腳本之後傳送到其它 Client 端。各個 Client 端再依照腳本加以播放，若有語音部份則由 TTS 功能將之復原成語音播放。同樣的，這種帶語音標籤的 Broadcasting Script 是不會傳到主體(Us)的。如下圖:

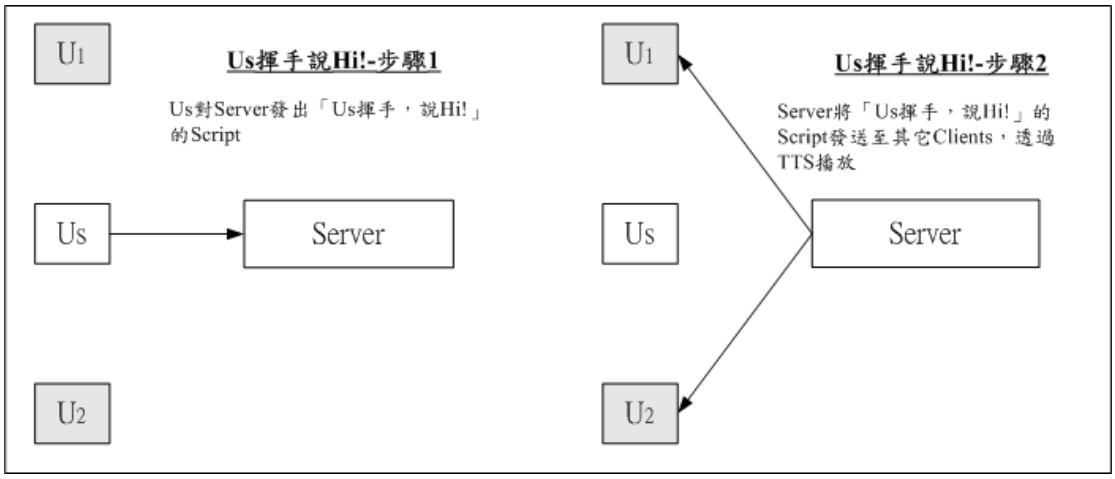

圖 3.2：由人類扮演的 Client 如何將動畫及語音以 Script 的方式傳送給他其它 Clients。人的語音可以由語音辨識(ASR)或鍵盤輸入，而傳到 U1,U2 後可以語音合成(TTS)或文字輸出。

3.2 多人虛擬環境中的對話管理

3.1 中我們針對虛擬環境中只有二個人且沒有對話的情況下腳本的傳播方式。在這一節中我們會針對多人虛擬環境中對話傳播方式加以討論。

3.2.1 多人虛擬環境中的對話管理架構

在本研究中我們採用 VoiceXML 的對話模型來表現對話過程。VoiceXML 主要運作是基於 HTTP 協定與 HTML 的 From 來表達對話概念，將對話看成是一連串 HTTP Request 與 HTTP Response 的結合。

使用 VoiceXML 對話架構進行對話時必須有「對話文件伺服器(Document Server)」的輔助。對話流程與邏輯判斷則委由對話文件伺服器上的伺服器端動態腳本機制(Server-side Scripting)來處理。依據使用者所送來的回應，對話文件伺服器再動態地回應下一段對話腳本(Dialog Script)。

如圖 3.3 步驟 1 所示，Us 一開始透過某個 HTTP GET 向對話文件伺服器(在圖中以 Document Server 表示)要求某個 URL。對話文件伺服器收到後，根據此 URL，可能直接傳回靜態 Script，或在伺服器端執行某些腳本，動態產生下一段對話腳本後才回傳給 Us(步驟 2)。Us 收到回傳後，即可解譯並播放這一段互動給使用者看並等待其回應(步驟 5)。使用者回應後，Us 會將回應 POST 回文件伺服器的另一個 URL，重複進行直到中斷對話為止(步驟 6~10)。所以整個對話過程中，其實客戶端都是在和文件伺服器互動。

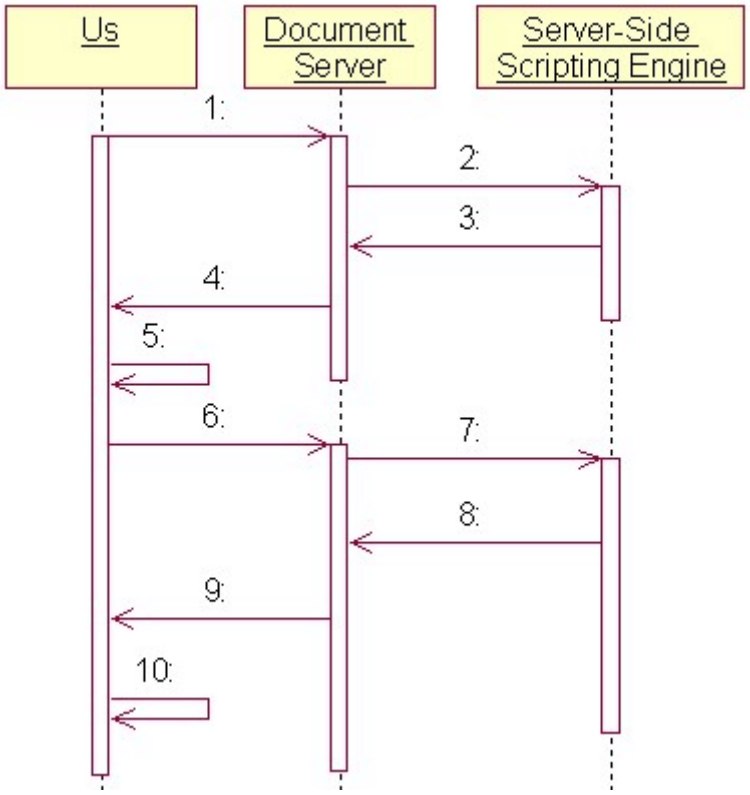

圖 3.3　VoiceXML 的對話方式。圖中描述 Us 與 Document Server 的互動方式。其行為可類比為 Web Browser 與 Web Server 的互動，對話腳本由 Server-Side Scripting Engine 動態產生。

由以上的解析可看出，這個機制要模擬虛擬環境中二個虛擬人物的對話，並非十分直覺。因為所有的對話腳本(Dialog Script)皆須向對話文件伺服器取得，另外，VoiceXML 機制是基於 HTTP 協定，而 HTTP 協定和大部份的虛擬環境所使用的協定並不相同。

我們以圖 3.4 來說明這種不自然的狀況。對話初始時，系統(Ss)會先透過廣播方式將第一份對話腳本的 URL 傳送給使用者(Us) (如圖 3.4，步驟 1)。這個 URL 中包含了對話文件伺服器(Document Server)的位置。使用者(Us)從此之後便依據這個 URL 向對話文件伺服器送出 HTTP 請求，伺服器依據此請求傳回下一份對話腳本，使用者(Us)方的 Client 程式此時會播放此腳本的內容給使用者看，再依據使用者回應向對話文件伺服器(Doc)送出下一個 HTTP 要求，週而復始直到結

束對話為止。(如圖 3.4，步驟 3~6)

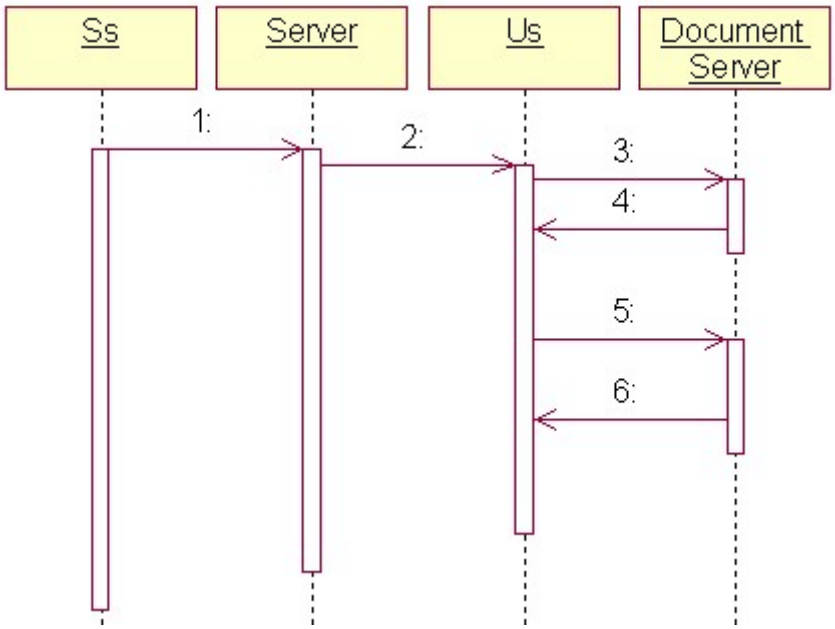

圖 3.4　多人環境中引用 VoiceXML 後所產生不自然對話的情況，Us 以為在和 Ss 對話，其實它只是不斷從 Document Server 取得腳本，和 Ss 反而無關。

在這種對話模型中出現最不自然的情況便是使用者(Us)會認為自己是在和系統(Ss)對話，但其實所有的對話內容都是由對話文件伺服器得來。真正的系統(Ss)在其對話過程中完全沒有參與。這種模式最大的問題在於，系統(Ss)一但送出對話文件伺服器的網址後便沒有辦法影響對話的內容與流程，對話實際的參與者只有對話文件伺服器與使用者(Us)，而對話中途所產生的任何結果與訊息，系統(Ss)也完全不知情。若對話途中，因網路產生錯誤或其它意外而中斷對話時，系統(Ss)也無法針對這些情況做有效的處理。

針對這些問題，我們提出另一種修正過的對話模型加以解決。這個模型是將對話文件伺服器(Document Server)要求與回應處理都委由系統(Ss)負責。如圖 3.5，步驟 1~2 所示，對話初始時，系統(Ss)會先到對話文件伺服器抓取第一份對

話腳本，並透過廣播方式將第一份對話腳本的 URL 傳送給使用者(Us)(步驟 3~4)。Us 在取得使用者的回應後，又再度透過 Ss 要求下一份腳本(步驟 5~7)。

這種做法可以讓對話文件伺服器完全獨立於虛擬環境之外。對於使用者端的 Client(Us)來說，他和系統端(Ss)的溝通方式是透過 Server 來轉送，而回應也是由系統端(Ss)送出。所以對使用者端來說，他和系統端(Ss)的對話本質上就是真的是和系統端(Ss)在做訊息的交換，比原來的模型合理。另外一個好處是所以的對話過程與結果系統端(Ss)都可以察覺，若有發生意外的網路錯誤也可以加以處理。

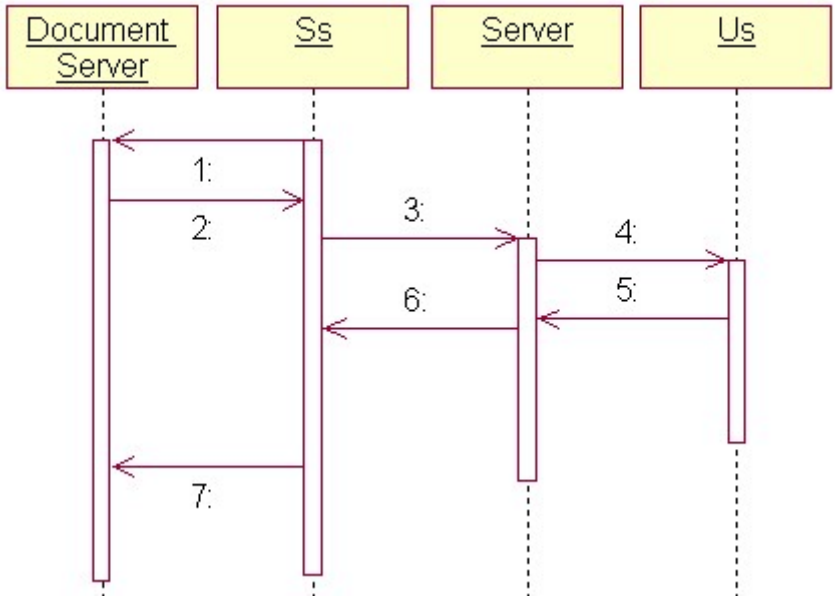

圖 3.5　改良後的對話運作方式，與對話文件伺服器(Doc)的溝通完全由系統(Ss)來處理。

要使此一改良過的對話模型正確地在多人虛擬環境下運作，我們尚須考慮多人交談時的競爭問題（例如現實生活中二個人在同一時間對 A 說話時，可能二個人講的話都無法被 A 所理解），所以我們在 3.2.2 中會使用「對話權利」的機制來解決這個問題。

3.2.2 對話權利(Dialog Lock)

以一般 HTTP 運作機制而言，同時會有多個 Web Clients 前來 Fetch HTML 文件回去，並呈現給使用者看。換句話說，VoiceXML 的對話文件伺服器，也能會有多個 VoiceXML Clients 同時 Fetch 文件回去，播給他的使用者看(並互動)。這代表著同時可能同時有多個人在和同一個人交談，但這種現象在虛擬環境中會造成不合理的現象(例如現實生活中二個人在同一時間對 A 說話時，可能造成二個人講的話 A 都聽不清楚)。

因此本研究限定同一時間同一個對話只有「二個人」的參與。當虛擬環境中任二個人物在交談時，雙方都不能再和別人對話。從實作觀點來看，這二個人的「對話權利」必須被鎖定起來，直到任何一方結束對話，才會通知對方解除這個鎖定。所以在所有的虛擬環境 Client 端程式都要有「對話權利」的控制機制，我們稱此為「Dialog Lock」。

接下來我們將說明為什麼本研究限定同一個時間同一對話只能有二個人參與。若仔細分析日常生活中所有對話的發生模式，我們會發現以下規則:

假設我們對他人說話為「輸出」
他們對我們說話為「輸入」
則合理的對話方式是「輸出」同時可 1 對 1~n，而輸入同時只能 1 對 1。

考慮 A 對 B、C 及 D 三人演說的情境，A 講的話(對 A 來講為輸出)，B、C、D 都可以聽到。假設現在演講完了 B、C 及 D 都要提問，一次也只能一個人出聲，若三人一起問，A 一定沒有辦法聽清楚任何一個人的問題。

套用到虛擬環境中，B 若要提問，就得先拿到 A 的「Dialog Lock」，才能與 A 交談，此時 C 和 D 都只是在旁聆聽。這時候交談中的 A、B 為主體，C、D 為旁觀者。假設 A 是系統扮演的，這時候 B 其實是一直接收並處理 Dialog Script。而 C 和 D 所收到的是「A 與 B 對話的實況轉播」，也就是 Broadcasting Script。而只有 A 在演講的時候，其實 B、C 與 D 收到的都是 A 所發出來的 Broadcasting Script，因為他們之中沒有任何二人有互動。

由於有 Dialog Lock 的機制，所以「啟始對話」和「結束對話」的機制就變得十分重要。因為「啟始對話」時要 Lock 雙方的「對話權利」，結束時又要 release。在本研究中我們假設*雙方都要有結束對話的權利*。換句話說，當二人中有一人不想再說了，就會導致對話的結束。

3.2.3 啟始對話的協定

「啟始對話」時，要檢查看看雙方是否都具有「對話權利」，若有一方的被鎖住，代表他正和別人對話，這時候「啟始對話」的要求就會被拒。

啟始對話有二種可能：*使用者(Us)主動啟始與系統(Ss)主動啟始*二種。在這裏以 Us 主動啟始的情況來做說明，因為實作上由 Us 主動較直觀，因為無論 Us 是否在 Ss 的可視範圍，可以想像只要按個鈕，Ss 就會走過來和 Us 講話。但若要讓 Ss 主動搭訕 Us 的話，必須在 Ss 這個 Client 寫程式實做啟始的動機，例如「當 Us 離我多近時，我主動問他要什麼服務」或「我要在某時主動拜訪某些人」等等。

假設由 Us 主動啟始對話，首先 Us 要先確認自己是否在對話的狀態(也就是「Dialog Lock」是否被鎖住)。若沒問題，則會透過 Server 傳送「Us 要求對話」

(Dialog Request from Us)的訊息給 Ss(如圖 3.6，步驟 1~3)。Ss 收到「Us 要求對話」的訊息後，也會先確認自己是否在對話的狀態(步驟 4)，若不在對話狀態，則傳送一個「Ss 同意對話」(Dialog Accepted from Ss)的訊息給 Us(步驟 5~6)，Us 收到後，便會將自己的 Dialog Lock 鎖住(步驟 7)。最後 Us 再向 Ss 發出一個「確認」訊息(步驟 8~9)。Ss 收到確認後就可以將自己的 Dialog Lock 鎖住(步驟 10)，並依據他專屬對話文件伺服器的 URL 去取回第一份對話腳本並傳送給 Us，開始對話過程(步驟 11~13)。反之，若 Ss 發現自己正在進行另一個對話，則必須傳送「拒絕對話」的訊息回去給 Us(如此 Us 的 Client 才能正確告知使用者要求失敗的原因)。

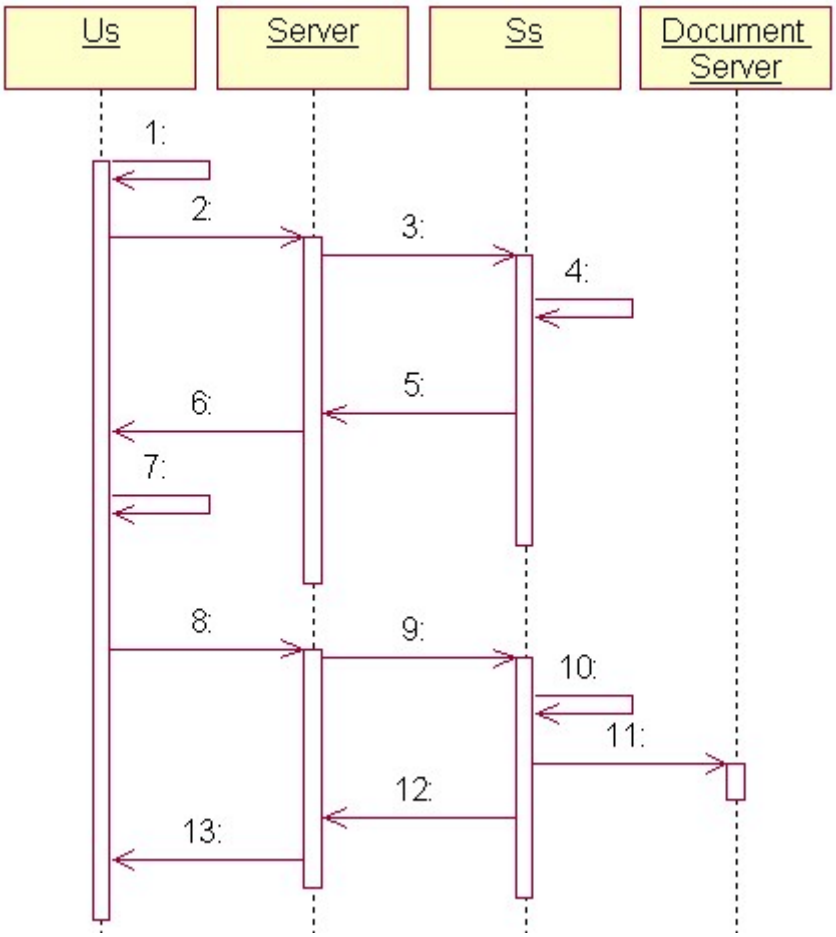

圖 3.6　Us 向 Ss 要求啟始對話

當雙方確認進入對話狀態後，就可以開始對話。此時 Us 將開始向使用者播放剛才 Fetch 到的 Script。使用者互動的結果，Us 將結果傳給 Ss，而 Ss 再將其透過 HTTP POST 傳回 Document Server 的另一個 URL，Document Server 再依此回應執行 Server Side Script，結果又會傳回另一份新的 Script 給 Ss，Ss 再回傳給 Us。於是又重複進行上述步驟，直到中斷對話為止。所以整個對話過程中，其實都是 Ss 在和 Document Server 在互動，以決定下一段對話。

3.2.4 多人環境中對話協議的同步問題

在多人環境中，以上這種方式會發生不同步的問題。例如使用者 Us 向 Ss 發出要求對話訊息後，Ss 此時沒有在對話，因此回傳「接受對話」的訊息，在 Us 還沒發出確認之前，另外有一個使用者 Ut 也發出了「要求對話」的訊息。此時因為 Ss 也不在對話狀態，所以也會回傳「接受對話」的訊息。如此一來在若二人同時向 Ss 要求對話(Dialog Request)時，Ss 會對二人都發出同意對話(Dialog Accepted)的訊息，於是二個人都認為自己在和 Ss 對話，而且都把自己鎖起來，此時 Ss 只會同意後要求的人(Ut)的對話，造成先發出要求的人(Us)發生錯誤狀態。

第二種同步的錯誤是使用者 Us 向 Ss 發出「對話要求」的訊息後，在 Ss 回傳前，又向另一個系統角色 St 要求對話，此時若 Us 與 St 趕在 Ss 回傳「接受對話」前就達成了對話協議，Ss 陷入停留在一直等待 Us 回應「確認對話」的錯誤狀態。

我們使用三個機制來解決這個問題:

1. **對話協商狀態(Dialog State)機制:**我們將整個協商對話的過程以三個不同的

狀態(State)來表示。

2. **同步(Synchronize)機制:**指出 Client 端實作時某些步驟必須一步完成。
3. **逾時(Timeout)機制:**為了預防鎖死或無限制等待，在某個狀態(State)下停留逾時後，就會回到原始的狀態。

每一個 Client 端都在參與對話的過程都以「Not in Dialog」、「In Dialog」與「Dialog Negotiation」等三個不同的狀態(State)來表示(如圖 3.7)。Client 程式啟動時預設為「Not in Dialog」狀態。在任一時間無論接收或傳送「要求對話」訊息時，Client 會馬上進入對話協議，也就是「Dialog Negotiation」的狀態。一個 Client 在同一個「Dialog Negotiation」狀態只能和一個對象做對話協商。每一個 Client 的「Dialog Negotiation」根據系統設定都會有一定的逾時(Timeout)時間，若超過這個時間還沒達成協商，則會放棄對話協商，自動回到「Dialog Negotiation」的狀態。另外，從系統 Ss 接到使用者的對話要求時，驗証自己的 Dialog Lock 到送出「Dialog Accepted」訊息必須在一個步驟內完成，這部份的功能可以使用語言的同步機制加以實作。

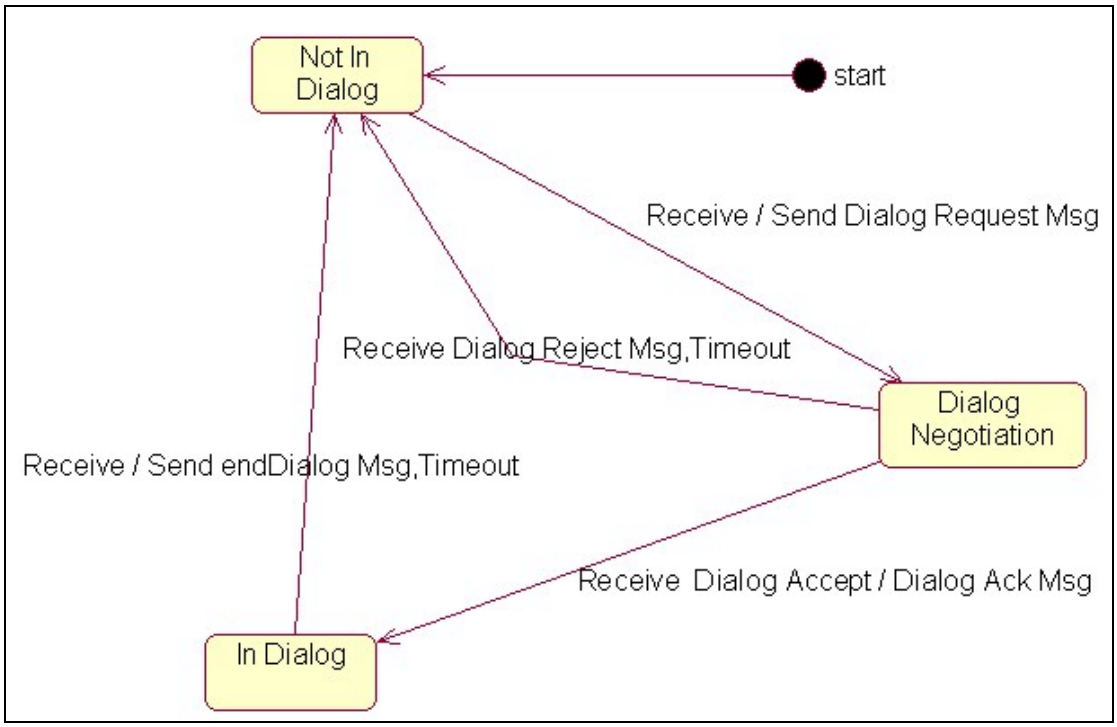

圖 3.7　Client 的三個對話狀態(Dialog State)

使用這種機制的除了能解決錯誤狀態的問題之外，另一個好處是無論是使用者(Us)所扮演的 Client 或系統(Ss)所扮演的 Client 都實作同一種對話狀態模型，簡化了系統設計的複雜度。

3.2.5 結束對話的協定

結束 Us 與 Ss 的對話也可能有二種情況：「正常結束」與「對話途中使用者主動要求結束」。

在一般的對話情況中，Ss 會依照 Us 傳送過來的 Script 中的指示，fetch 下一份 script 後再回傳，如此不斷重複，直到 Script 沒有要求另一份 Script 時，對話會自動終止。由於 Script 是在 Us 上執行，故 Us 可以得知對話結束時機。Us 一知道對話結束後，會馬上解除自己的 Dialog Lock，並發送一「與 Us 對話結束」的訊息給 Ss(如圖 3.8，步驟 1~3)。在 Ss 收到後，也會解除自己的 Dialog Lock。

此時便解除了二者的對話狀態(如圖 3.8，步驟 4)。

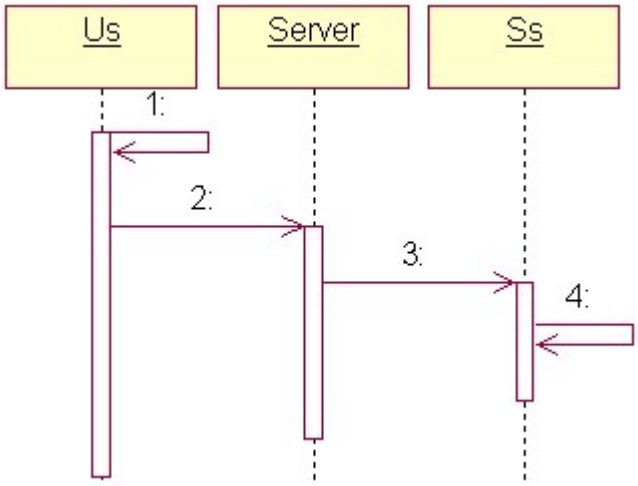

圖 3.8　正常結束對話

另外一種屬於對話途中，臨時想中斷對話的情況。可能的情況是使用者(Us)忽然不想跟系統扮演的角色(Ss)對話，而按下類似「強迫中止對話」的按鍵(在 VoiceXML 中可類比為使用者在對話過程中忽然掛電話)。在這種情況下，Us 的 Client 程式必須在使用者下達強迫中止對話的指令後，解除自己的 Dialog Lock，並將中止對話的訊息傳送至 Ss，所以這部份和自然結束對話的過程是相同的。

3.3 多人虛擬環境中的對話傳播

在典型的多人虛擬環境裏，我們除了要求進行對話的 client 必須正確下傳並播放 Script 之外，也要顧及現在其它旁觀者的感受。

做使用 XAML-V 腳本進行動作的角色，會將自己所做所為以 Script 的方式透過 Server 告知其它 Client，我們定義這種發出 Script 訊息者稱為*主體(Subject)*，其它接收端為*旁觀者(Observer)*。在對話的模式中，我們擴充了此一觀念，將進

行對話的二者定義為主體，其它觀察這二人對話者稱為旁觀者。

做如此區分的主因是參與對話的人(主體)所得到的 Script 通常和其它人(旁觀者)所得到的 script 不同。例如參與對話的 Us 通常得到的 Script 含有互動式的「<Form>」標籤(Us 和 Ss 互動)，而此時其它人所收到的大部份都是純播放用的「<prompt>」標籤(用來播放 Us 說了什麼，Ss 又說了什麼的 Script)。若將主體與旁觀者一視同仁的話，就好像所有的 Client 都同時在跟 Ss 對話，會造成不合理的現象。

所以在此必須將這二種功能不同的 Script 加以區分。我們稱主體間對話用的 script 稱為 *Dialog Script*，而其之間的對會透過 *Broadcasting Script*「實況轉播」給其它旁觀者看。

3.3.1 多人虛擬環境中對話進行方式

假設參與對話的主體為 Ss 與 Us，S_1，S_2，U_1，U_2 為旁觀者，其中 Si(i=1,2)為系統扮演，Ui(i=1,2)為人類扮演。一開始必須啟動對話（對話啟動的細節詳見 3.2.3），Us 與 Ss 啟動對話後， Ss 會去 Document Server 取得第一份 Dialog Script 並回傳給 Us。從 3.2 的討論我們知道，Ss 從 Document Server 上取得的文件，本質上是描述 Ss 向 Us 講的話或問的問題。所以「實況轉播時」時，Ss 的 Client 程式必須同時負責根據 Dialog Script，產生一份 Broadcasting Script，並透過 Server 向其它旁觀者發送(如圖 3.9，步驟一) 。

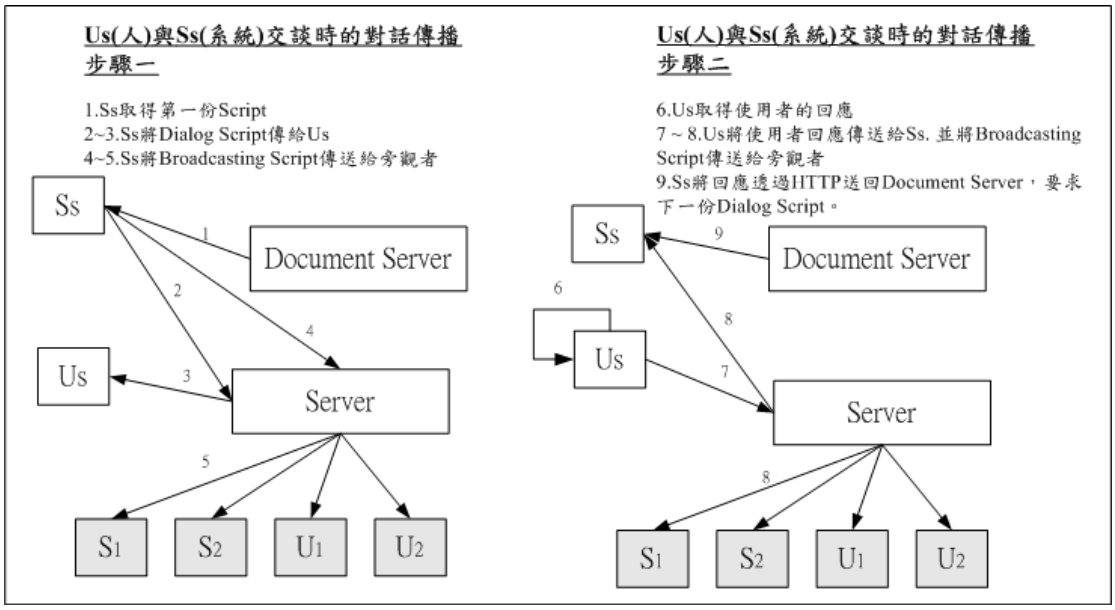

圖 3.9　多人環境中對話腳本的傳播

Script 開始在 Us 上執行後，Us 的使用者可能會對此 Dialog Script 產生一些回應(如回答一個問題)，依據 3.1 節所描述的模型，此時 Us 會將回應透過 Ss 傳送回對話文件伺服器。不過在傳送之前，Us 也需負責依據此回應產生合適的 Broadcasting Script 透過 Server 傳送給其它旁觀者。(如圖 3.9，步驟二)之後的對話過程則都是依這種方式進行。

本章主要針對 VoiceXML 對話模型在多人環境中的不足之處，發展一系列的觀念希望能對此模型加以擴充與修正。首先我們先探討了多人環境中動畫及對話腳本的傳播方式，接著針對 VoiceXML 在多人環境中所造成的不自然情況做了修正。

在多人環境中必須處理許多在單人環境中不會發生的問題，例如當二個人在同一時間對 A 說話時，可能造成二個人講的話 A 都聽不清楚，所以我們希望同時間內只限二個人參與一段對話，因此發展出對話權利(Dialog Lock)、對話狀態(Dialog States)、同步(Synchronize)與逾時(Time out)等觀念並將之應用在啟始與結

束對話的協議中，我們將整個對話過程分為「Not in Dialog」、「In Dialog」與「Dialog Negotiation」等三個不同的狀態(State)，來完成啟始與結束對話的協議。在下一章我們會針對本章所探討的機制以 XAML-V 加以實現，並在第五章實際以 Java 語言在 IMNet 多人虛擬環境上加以實作，以驗証在本章所討論的觀念與機制。