

5

開發高等分散 式多層應用系統

現在我終於瞭解了 MIDAS 的原理和它的奧妙，但是我如何能夠在多層應用系統中使用我對於 MIDAS 的瞭解以便實作出先進的多層應用系統呢？有那些技巧是屬於高等的 MIDAS 技術？使用 MIDAS 到底可以開發出什麼樣的多層應用系統。

本章重點

- 魚與熊掌兼得之法
- 開發具備容錯能力的多層應用系統
- 具備容錯能力的資料應用程式伺服器
- 開發具備平均負荷能力的應用系統 TDCOMConnection
- 基本的安全強固的應用系統
- 進階的安全強固的應用系統
- 開發使用 COM/DCOM 技術的安全強固應用系統

在前一章中討論了 MIDAS 的技術原理，現在你應該瞭解 MIDAS 的運作方式，在本章中將會根據我們對於 MIDAS 的知識進而開發一些高級的分散式多層應用系統。這些高級的技巧包括了如何使用 MIDAS 3.0 無狀態物件，如何在 MIDAS 無狀態物件中維護應用程式的狀態資訊，如何開發具備容錯能力以及平均負荷的應用系統等。這些高級的技巧對於一個安全強固的應用系統都是非常重要的，但是要讓應用系統具備這些高等的功能則必須有賴程式師對於 MIDAS 有徹底的瞭解，所以這是為什麼本章的內容是在討論 MIDAS 章節之後。

除了一般的分散式多層應用系統之外，在『實戰 Delphi 5.x-分散式 Web 應用系統篇』中會介紹如何使用 Delphi 5 開發 Internet/Intranet 以及電子商務的應用系統，在這些系統之中使用無狀態的 MIDAS 物件更為重要，但是這些應用系統也需要維持用戶端應用程式的資料狀態，因此這讓程式師陷入了兩難，因為要開發具延展性的 Internet/Intranet 或是電子商務應用系統必須搭配 Web Server，MTS，CORBA 伺服器等，這些中介軟體伺服器又要求企業物件最好是無狀態物件，那麼我們到底該如何克服應用程式需要維持資料的連貫性，但是又要讓企業物件是無狀態的呢？這些問題都會在本章中討論，讓你瞭解如何同時使用無狀態物件，又能夠開發出具備資料連貫性的應用系統，讓你的應用系統不必犧牲任何的功能，卻又能夠擁有高度的延展性，讓你開發的系統就是比別人的系統表現的更好。

5-1 魚與熊掌兼得之法

從前面的章節中我們瞭解了 MIDAS 3.0 在內定上是無狀態的，所以由 Delphi 5 建立的應用程式伺服器應該是無狀態物件。但是當用戶端應用程式把 TClientDataSet 元件的 FetchOnDemand 特性值設定為 True 要求 TClientDataSet 自動以分段的方式存取資料時，此時它連結的 TDataSetProvider 便又自動成為一個狀態物件，因為它必須幫助用戶端應用程式維護狀態資訊(例如目前資料表的游標位置)。

使用狀態物件本身沒有什麼不好，只要你的應用程式伺服器中包含的每一個遠端資料模組都服務一個固定的用戶端應用程式便沒有什麼問題。但是在分散式多層應用系統中，如果你想搭配 MTS 或是其他的中介軟體一起使用時，或是你使用 MIDAS 實作一個服務 Internet/Intranet 上使用者的應用系統，那麼你的應用程式伺服器便非常的沒有延展性。請想一想，當你使用 MIDAS 在 Internet/Intranet 應用系統中的時候，如果每一個使用者使用瀏覽器連結到你的系統時，你的應用程式伺服器必須一直維持在記憶體之中服務這個使用者，那麼當有數百個使用者同時使用你的系統時，應用程式伺服器便撐不住了。或是像 MTS 這種使用 pooling 技術的中介軟體，一但應用程式伺服器必須維護狀態資訊，那麼 MTS 便無法釋放應用程式伺服器之中的 COM 物件，而無法為這些 COM 物件提供 pooling 的好處，這也就失去了使用 MTS 的一個重要的因素-pooling 技術。

因此如果我們想使用 MIDAS 實作 Internet/Intranet 或是電子商務應用系統時，那麼最好是讓應用程式伺服器在服務了用戶端要求之後，能夠立刻的被重複使用讓其他的使用者能夠存取這個應用程式伺服器提供的服務。如此一來你的應用程式伺服器便具備了高度的延展性，並且能夠使用在 Internet/Intranet，電子商務，或是 MTS 等的中介軟體之中。

從上面的討論中我們似乎陷入了兩難，因為我們一方面需要延展性，又希望多層應用系統能夠維護資料的連貫性，例如我們希望能夠在用戶端應用程式的 TDBGrid 或是表格中看到連續的資料。這代表應用系統要有延展性就需要使用無狀態物件，要維護資料的連貫性就需要狀態物件，那麼我們該如何解決這些衝突的要求呢？

事實上我們只要使用一點技巧就可以同時結合無狀態物件和狀態物件的好處，程式師可以使用內定的 MIDAS 無狀態物件，但是卻可以維護一些狀態資訊。下面討論的技巧可以使用在非常多的場合中，除了多層的資料庫應用系統之外，也特別適合使用在 MTS 之中。在實作展示這個技巧的範例之前，先讓我們討論這個技巧使用的觀念。

請想一想，如果你在多層應用系統中瀏覽資料的話，那麼當你設定 TClientDataSet 元件的 FetchOnDemand 為 True，並且設定 PacketRecords 為 10，那麼當你要瀏覽第 11 筆資料時 TClientDataSet 會向它連結的 TDataSetProvider 要求傳送下一個資料封包的資料，但是 TDataSetProvider 如何知道下一個資料封包是什麼？因此 TDataSetProvider 會幫 TClientDataSet 維護資料表的游標位置，那麼當 TClientDataSet 要求下一個資料封包時，TDataSetProvider 就根據目前的游標位置來存取下一個資料封包的資料，然後再調整相對的游標位置。

這個工作過程事實上可以由程式師在用戶端應用程式輕易的幫助 TDataSetProvider 完成，而不需要 TDataSetProvider 維護這些資訊，如此一來就可以讓應用程式伺服器成為無狀態物件。能夠達成這樣的目標是因為 MIDAS 3.0 多增了一些事件處理函式可以讓程式師控制，也增加了 TClientDataSet 和 TDataSetProvider 交換資訊的機制，因此可以讓用戶端應用程式控制應用程式伺服器的行為並且維護它的狀態資訊。

在下面的範例中將要實作一個存取資料的多層應用系統，在這個範例中 TClientDataSet 的 FetchOnDemand 特性值是設定為 False 的，因此用戶端應用程式必須自己向應用程式伺服器取得隨後的資料。由於在這個範例中的 TDataSetProvider 是無狀態物件，所以它可以被許多的用戶端同時使用，當不同的用戶端應用程式需要使用這個 TDataSetProvider 元件存取不同的資料時，只需要傳遞每一個用戶端應用程式正確的游標位置，再讓 TDataSetProvider 搜尋到正確的資料，然後傳回下一個包含正確資料的資料封包即可。問題是用戶端應用程式如何和 TDataSetProvider 合作維護正確的游標資訊呢？下面的圖形即說明了這個流程：

2. 在 BeforeGetRecords 事件 處理

函式中檢查是否有搜尋欄位鍵值，把上一次資料封包的最後一筆的鍵值傳遞給 TDataSetProvider

3. 在 BeforeGetRecords 事件處理函

式中根據 TClientDataSet 傳遞來的 鍵值搜尋資料，並且回傳資料封包

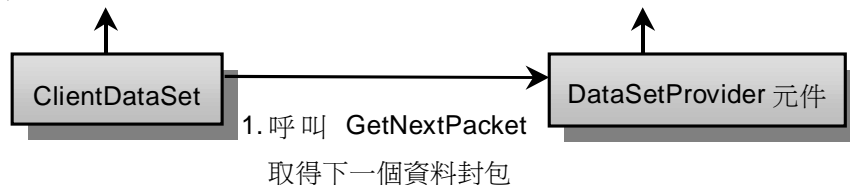


圖 5-1 用戶端應用程式維護狀態資訊的流程

在 Delphi 5 中，當用戶端應用程式呼叫 `GetNextPacket` 向應用程式伺服器取得資料時，在 `TClientDataSet` 會觸發 `BeforeGetRecords` 事件處理函式。在這個事件處理函式中 `TClientDataSet` 可以把上一次存取資料封包的最後一筆鍵值傳遞給 `TDataSetProvider` 元件。當新的資料封包傳送到用戶端應用程式之後，`TClientDataSet` 的 `AfterGetRecords` 事件處理函式便會觸發，在這個事件處理函式中你便可以儲存最新資料封包的最後一筆鍵值，以便下一次呼叫 `GetNextPacket` 時使用來正確的設定 `TDataSetProvider` 的游標位置。

現在就讓我們使用這些知識來實作一個可以由用戶端應用程式控制存取資料的範例。請建立一個新的專案，並且設計它的主表格如圖 5-2 所示。這個範例使用 `TSocketConnection` 連結到應用程式伺服器，而 `TClientDataSet` 元件的 `FetchOnDemand` 設定為 `False`，同時它的 `PacketRecords` 設定為 10。因此在範例程式一執行時用戶端只有 10 筆資料，而且無法自動的從應用程式伺服器取得隨後的資料。在表格中有一個『More...』按鈕，當使用者希望繼續瀏覽隨後的資料時，可以點選這個按鈕。那麼用戶端應用程式便會呼叫 `GetNextPacket` 向應用程式伺服器取得下一個資料封包。

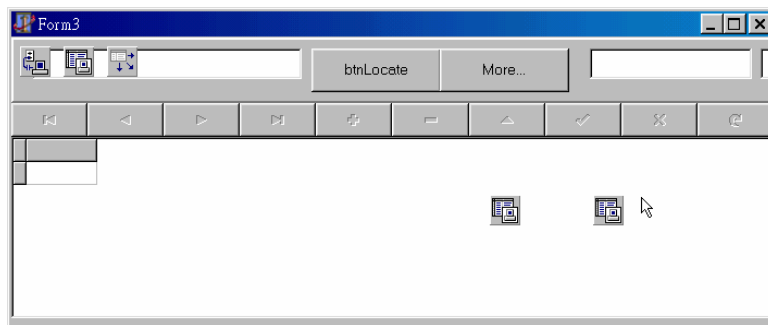


圖 5-2 範例用戶端應用程式主表格

現在就讓我們看看這個範例應用程式如何維護狀態資訊，以便依據這些狀態資訊向應用程式伺服器取得正確的資料。下面是『More...』按鈕的 `OnClick` 事件處理函式：

```
procedure TForm3.btnMoreClick(Sender: TObject);
```

```
var
  lStart, lEnd : Integer;
  abk : TBookmark;
begin
  lStart := GetTickCount;
  try
    if (not bTrueEOF) then
      begin
        try
          aBK := ClientDataSet1.GetBookmark;
          if (ClientDataSet1.GetNextPacket <
              ClientDataSet1.PacketRecords) then
            bTrueEOF := True;
            ClientDataSet1.Last;
            vOwnerData := ClientDataSet1.FieldByName('NUM').Value;
            ClientDataSet1.GotoBookmark(aBK);
            ClientDataSet1.Next;
          finally
            ClientDataSet1.FreeBookmark(aBK);
          end;
        end;
      except
        on Exception do;
        end;
      edtRecNo.Text := IntToStr(ClientDataSet1.RecordCount);
      lEnd := GetTickCount;
      edtRecNo.Text := IntToStr(ClientDataSet1.RecordCount);
      edtTime.Text := FloatToStr((lEnd - lStart) / 1000.0) + '秒';
    end;
```

在這個事件處理函式中首先先使用書籤保留目前游標的位置，然後呼叫 TClientDataSet 的 GetNextPacket 取得下一個資料封包，如果 GetNextPacket 回傳的筆數小於 TClientDataSet 的 PacketRecords 就代表已經到了資料表的結尾。在呼叫完 GetNextPacket 之後，先儲存這次資料封包最後一筆的鍵值，然後就回到原先的游標位置，再跳到下一筆，這樣就可以維持資料正確的相關位置。從上面的程式碼中似乎看不出來有什麼玄機，這個範例最重要的地方是它的 BeforeGetRecords 事件處理函式。

在用戶端應用程式中最後一個步驟便是 TClientDataSet 的 BeforeGetRecords 事件處理函式，當 TClientDataSet 要向應用程式伺服器取得下一個資料封包時，不管是呼叫 GetNextPacket 或由 FetchOnDemand 特性，都會觸發這個事件處理函式。所以在這個範例中，我們需要在這個事件處理函式中告訴 TClientDataSet 連結的 TDataSetProvider 取得正確的資料封包，因此在這個事件處理函式中我們把上一次存取的資料封包的最後一筆記錄的鍵值儲存在 vOwnerData 變數之中，最後再把游標的位置歸位。在 Delphi 5 中程式師只需要設定這個事件處理函式的 OwnerData 參數即可，MIDAS 會自動的把這個參數值傳送給 TDataSetProvider 元件。

```

procedure TForm3.ClientDataSet1BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  if ((VarIsEmpty(vOwnerData)) or (VarIsNull(vOwnerData))) then
    vOwnerData := GetKeyFieldValue(Sender);
  OwnerData := vOwnerData;
end;

```

在上面的程式碼中，BeforeGetRecords 會檢查上次儲存的鍵值是否為空白，如果是的話，就呼叫 GetKeyFieldValue 以取得目前在 TClientDataSet 中最後一筆資料的鍵值。至於 GetKeyFieldValue 的程式碼如下：

```

function TForm3.GetKeyFieldValue(Sender : TObject): OleVariant;
var
  CurRecord: TBookmark;
begin
  try
    //保持上一次存取資料的最後一筆的資訊
    with Sender as TClientDataSet do
      begin
        CurRecord := GetBookmark; { 儲存目前記錄的位置 }
        try
          Last;
          Result := FieldByName('NUM').Value;
          GotoBookmark(CurRecord); { return to current record }
        finally
          FreeBookmark(CurRecord);
        end;
      end;

```

```
    end;  
  except  
    on Exception do;  
  end;  
end;
```

GetKeyFieldValue 根據傳入的 TClientDataSet，跳到最後一筆，儲存鍵值，最後再回到 TClientDataSet 原來的游標位置。

前面的程式碼是在用戶端應用程式中需要撰寫的，而在應用程式伺服器中程式師必須為 TDataSetProvider 元件撰寫它的 BeforeGetRecords 事件處理函式。在這個事件處理函式中，TDataSetProvider 元件利用用戶端傳遞來的鍵值在它連結的資料表中搜尋這筆資料，把游標移動到上次最後一筆被存取的資料位置即可。在下面的程式碼中你可以看到 TDataSetProvider 元件直接呼叫它連結的資料集元件的 Locate 方法移動游標到鍵值的記錄。

```
procedure TldsDemoServer.DataSetProvider1BeforeGetRecords(Sender:  
TObject; var OwnerData: OleVariant);  
begin  
  with Sender as TDataSetProvider do  
  begin  
    DataSet.Open;  
    DataSet.Locate('NUM', OwnerData, []);  
    DataSet.Next;  
  end;  
end;
```

當然，如果在用戶端應用程式需要使用多個欄位值做為搜尋資料的標準，那麼在 TClientDataSet 的 BeforeGetRecords 事件處理函式中程式師可以建立一個 Variant 陣列，在這個 Variant 陣列中置入所有的欄位值，再把這個 Variant 陣列傳遞給 TDataSetProvider，最後在 TDataSetProvider 的 BeforeGetRecords 事件處理函式中再從 Variant 陣列中取出所有的搜尋欄位值，再呼叫 Locate 以多個欄位來搜尋資料即可。

完成了在用戶端和伺服器端必要的程式碼之後，讓我們執行這個應用系統看看它是否能夠正確的取得資料。下面的畫面是用戶端應用程式執行的情形：



圖 5-3 範例程式執行畫面

範例程式一開始執行時顯示了它取得了 10 筆的資料，我們捲動到最後一筆發現它的鍵值是 000011。接著點選『More...』按鈕向應用程式伺服器取得下一個資料封包，從下圖中你可以看到範例程式果然正確的取得了下一個資料封包的資料，同時範例程式也顯示此時在用戶端應用程式擁有 20 筆資料。

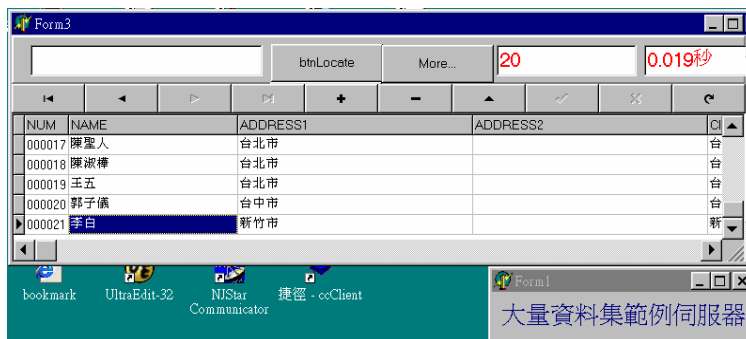


圖 5-4 範例程式取得下一個資料封包

為了證明這個技巧可以正確的向 TDataSetProvider 取得資料，請回到 Delphi 5 中繼續的修改這個範例，請在表格中加入另外一個 TClientDataSet，設定這個新的 TClientDataSet(ClientDataSet2)元件的 RemoteServer 為表格中的 TSocketConnection，同時設定它的 ProviderName 特性值和剛才表格中 ClientDataSet1 相同的 TDataSetProvider 元件，最後設定這個新的 TClientDataSet 元件的 PacketRecords 為 30，讓它和 ClientDataSet1 使用不同的

PacketRecords，這樣可以讓這兩個 TClientDataSet 元件在存取資料會使用不同的資料區間。

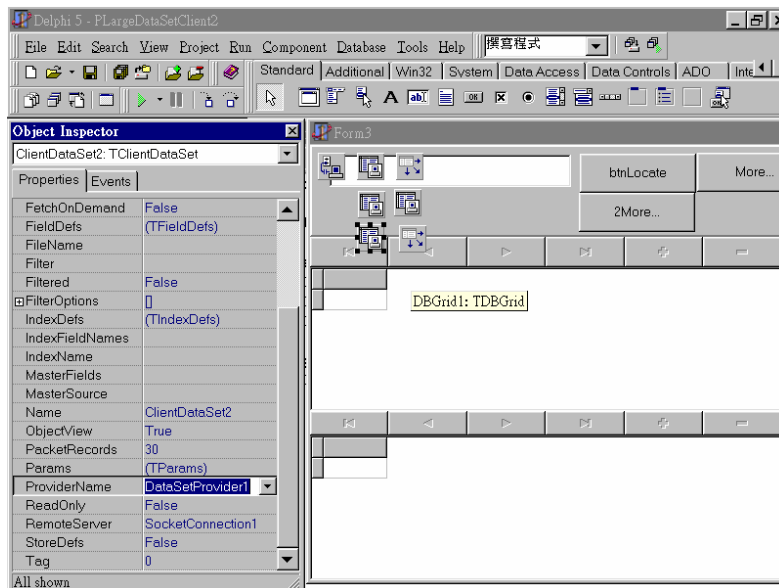


圖 5-5 修改過的主表格，使用額外的 TClientDataSet 元件

由於現在在範例程式的主表格中有兩個 TClientDataSet 使用連結相同的 TDataSetProvider 來存取元件，所以當 ClientDataSet1 第一次存取資料時，它會存取資料表中前 10 筆的資料，但是當 ClientDataSet2 存取資料時，它會存取 30 筆的資料。所以如果我們沒有使用這種技巧，那麼當 ClientDataSet1 存取下一個資料封包時，它應該會從第 31 筆資料開始存取，而不會從正確的第 11 筆存取，這是因為資料表的游標位置被 ClientDataSet2 改變了。

表格中的『2More...』按鈕就像『More...』按鈕的功能一樣，是讓 ClientDataSet2 呼叫 GetNextPacket 來取得資料的，它使用了和『More...』幾乎一樣的程式碼，也定義了 ClientDataSet2 的 BeforeGetRecords 事件處理函式：

```

procedure TForm3.ClientDataSet2BeforeGetRecords(Sender: TObject;
  var OwnerData: OleVariant);
begin
  if ((VarIsEmpty(v2OwnerData)) ) then

```

```
v2OwnerData := GetKeyFieldValue(Sender);
OwnerData := v2OwnerData;
end;

procedure TForm3.btn2MoreClick(Sender: TObject);
var
  lStart, lEnd : Integer;
  abk : TBookmark;
begin
  lStart := GetTickCount;
  try
    if (not bTrueEOF2) then
      begin
        try
          aBK := ClientDataSet2.GetBookmark;
          if (ClientDataSet2.GetNextPacket <
            ClientDataSet2.PacketRecords) then
            bTrueEOF2 := True;
          ClientDataSet2.Last;
          v2OwnerData := ClientDataSet2.FieldByName('NUM').Value;
          ClientDataSet2.GotoBookmark(aBK);
          ClientDataSet2.Next;
        finally
          ClientDataSet2.FreeBookmark(aBK);
        end;
      end;
    except
      on Exception do;
    end;
  end;
  edtRecNo.Text := IntToStr(ClientDataSet2.RecordCount);
  lEnd := GetTickCount;
  edtRecNo.Text := IntToStr(ClientDataSet2.RecordCount);
  edtTime.Text := FloatToStr((lEnd - lStart) / 1000.0) + '秒';
end;
```

現在執行這個修改過的範例程式，從圖 5-6 中可以看到在範例程式一開始時這兩個 TClientDataSet 都正確的存取到了資料，但是 ClientDataSet1 是存取 10 筆，而 ClientDataSet2 則是存取 30 筆。

5-12 實戰 Delphi 5.0-分散式多層應用系統篇



圖 5-6 修改過的範例程式執行的畫面

現在點選表格中的『More...』按鈕，從圖 5-7 中可以看到 ClientDataSet1 的確存取到了包含從第 11 筆到 20 筆的資料，雖然 ClientDataSet1 和 ClientDataSet2 同時使用了相同的 TDataSetProvider 元件，但是由於使用本小節討論的技巧，所以這兩個 TClientDataSet 並不會衝突，也證明了 TDataSetProvider 元件在內定上是無狀態物件，並不會為任何連結到它的 TClientDataSet 元件維護任何的資訊。

這種特性讓包含 TDataSetProvider 的遠端資料模組成為非常適合使用在 MTS 等中介軟體之中，因為每一個遠端資料模組可以同時服務數個不同的用戶端應用程式，這可以讓遠端資料模組能夠充分的使用物件 pooling，資料庫 pooling 等增加延展性和執行效率的技術。



圖 5-7 修改過的範例程式執行的畫面

由於本小節討論的技巧非常的有用，所以在『實戰 Delphi 5.x-分散式 Web 應用系統篇』一書中將會經常的使用這個技巧來增加分散式 Internet/Intranet，或是電子商務應用系統的延展性。

5-2 開發具備容錯能力的多層應用系統

在一個執行關鍵作業的多層應用系統中，系統的穩定度是非常重要的。尤其是當用戶端應用程式正在執行一些重要的程式時，絕不會希望因為應用程式伺服器的故障而造成多層應用系統無法執行。在第 4 章中介紹了 MIDAS 在多層應用系統中的容錯能力，但是程式師在 Delphi 中到底如何能夠實作出具備容錯能力的多層應用系統呢？答案就在 Delphi 5 中的一個元件『TSimpleObjectBroker』。這個元件提供了基本的容錯能力和平均負荷的能力。程式師可以使用這個元件實作出容錯能力的多層應用系統，此外程式師也可以繼承這個元件實作更複雜的容錯能力。本小節將討論如何使用 Delphi 實作容錯能力的多層應用系統，稍後的章節將會討論實作平均負荷的技術。

基本上容錯能力就是讓應用程式伺服器在多個機器中執行，當用戶端應用程式執行時可以連結到任何一台機器中的應用程式伺服器要求服務。如果用戶端應用程式連結的應用程式伺服器發生任何問題而無法繼續執行時，用戶端應用程式可以立刻連結到其他機器中的應用程式伺服器繼續要求新的應用程式伺服器提供服務。

所以開發具備容錯能力的多層應用系統在伺服端的應用程式伺服器是沒有什麼不同的，但是在撰寫用戶端應用程式時程式師就必須配合 MIDAS 撰寫一些額外的程式碼來實作容錯能力。讓我們使用下面的範例來說明如何實作容錯能力的多層應用系統。

這個範例是開發一個應用程式伺服器，然後執行此應用程式伺服器在兩台機器之中，並且啟動一個用戶端應用程式連結到其中一個應用程式伺服器。當用戶端應用程式執行之後，我們關閉它連結的應用程式伺服器，那麼再於用戶端應用程式要求應用程式伺服器提供服務，但是由於用戶端應用程式連結的應用程式伺服器已經被關閉了，所以它必須有一種機制能夠知道這個狀況並且試

著連結其他機器之中的應用程式伺服器以要求服務。現在就使用這個範例實作應用程式伺服器和用戶端應用程式。

1. 開發應用程式伺服器：容錯能力的應用程式伺服器和一般的應用程式伺服器沒有什麼不一樣，因為容錯能力必須由用戶端應用程式和 TSimpleObjectBroker 合作以提供應用程式伺服器的容錯功能。在這個範例中的應用程式伺服器只是一個連結到 Employee 資料表並且對外提供系統時間以及伺服器名稱的應用程式伺服器而已。下圖是這個應用程式伺服器輸出的特性以及實作這些特性的程式碼。

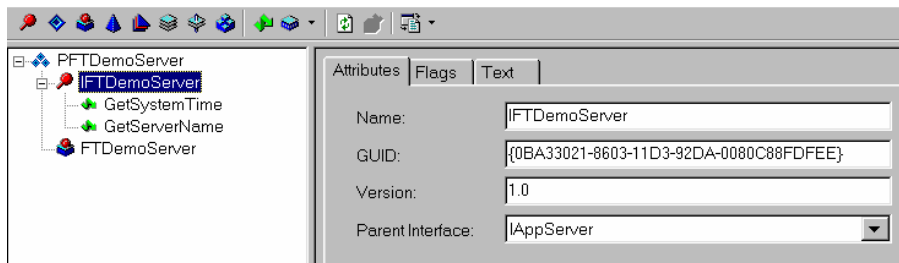


圖 5-8 應用程式伺服器輸出的特性

```

procedure TFTDemoServer.GetServerName(var vName: OleVariant);
begin
    vName := '容錯能力伺服器';
end;

procedure TFTDemoServer.GetSystemTime(var vTime: OleVariant);
begin
    vTime := DateTimeToStr(Now);
end;

```

2. 開發具備容錯能力的用戶端應用程式：要開發具備容錯能力的用戶端應用程式，程式師必須使用 TSimpleObjectBroker 元件。TSimpleObjectBroker 元件會維護一組能夠執行應用程式伺服器的機器，並且提供其中的機器名稱給 TDCOMConnection 或是 TSocketConnection 元件做為連結的遠端機器名稱。當 TDCOMConnection 或 TSocketConnection 連結的機器故障時，TDCOMConnection 或是 TSocketConnection 可以從 TSimpleObjectBroker 取得一個新的能夠執行應用程式伺服器的遠端機器，然後再連結這台新機器

以取得應用程式伺服器。在這個範例用戶端應用程式中，請先置入一個 TSocketConnection 元件，並且如下圖般放入其他的元件。此時在置入 TSimpleObjectBroker 元件之前，請先在物件檢視器中設定 TSocketConnection 要連結的應用程式伺服器名稱，也就是 TSocketConnection 的 ServerName 特性值。程式師可以藉由先設定 TSocketConnection 的 Host 特性為一台擁有應用程式伺服器的機器來先設定此 TSocketConnection 要連結的應用程式伺服器，然後再刪除剛才設定的 Host 或是 Address 的特性值。最後就可以設定 TSocketConnection 的 ObjectBroker 特性值為剛才加入的 TSimpleObjectBroker 元件。

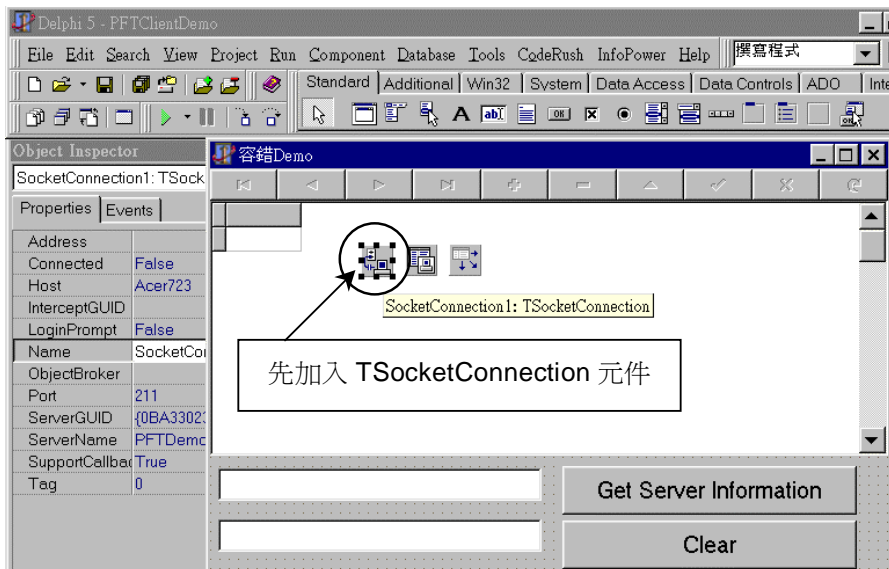


圖 5-9 建立容錯能力的用戶端應用程式

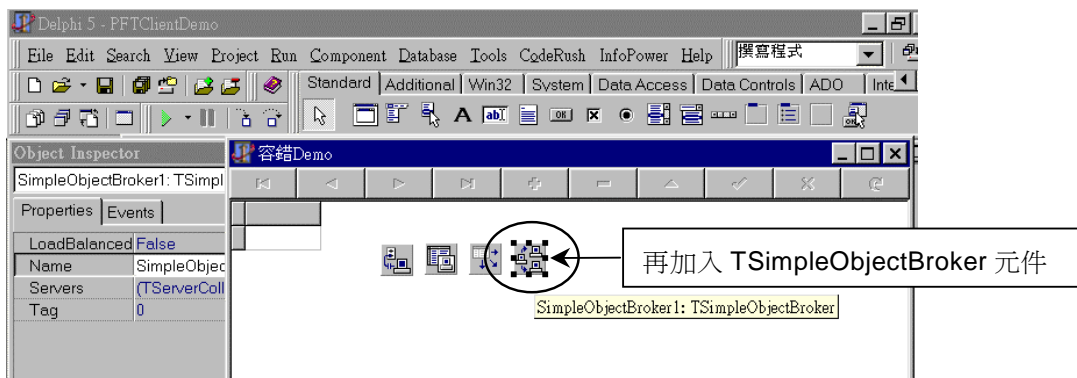


圖 5-10 使用 TSimpleObjectBroker 元件

3. 加入了 TSimpleObjectBroker 元件之後，就可以啟動 Servers 特性值編輯器加入能夠執行應用程式伺服器的機器。程式師可以點選物件檢視器中 Servers 特性，Delphi 會顯示一個『瀏覽電腦』對話盒，在這個對話盒中程式師可以加入所有可以執行應用程式伺服器的機器，如下圖所示：

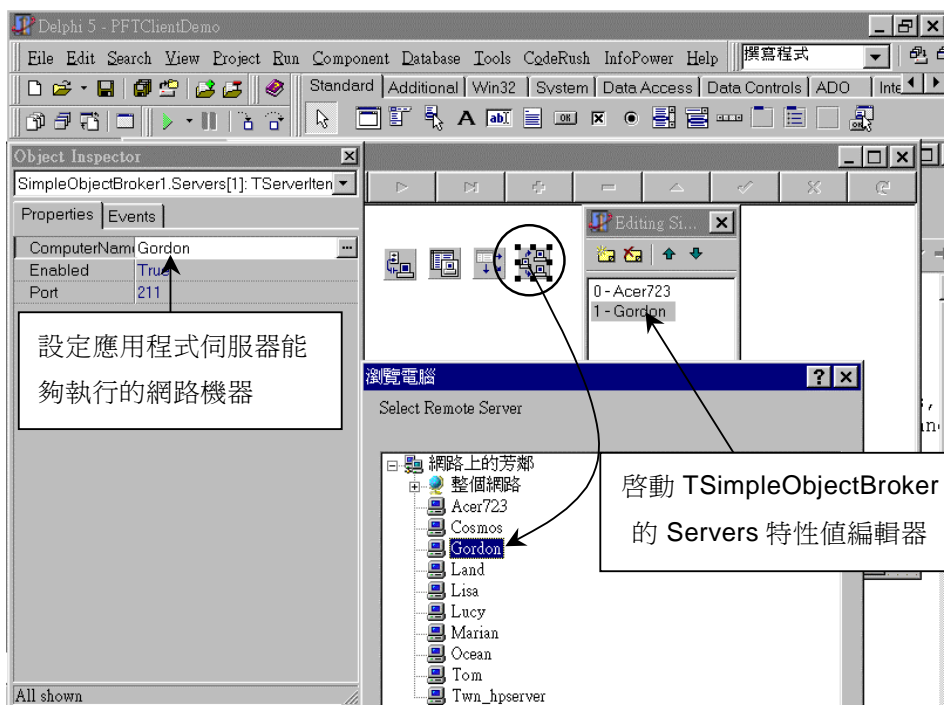


圖 5-11 使用 Servers 特性值編輯器加入可以執行應用程式伺服器的電腦

4. 在 TSimpleObjectBroker 之中定義的機器是所有能夠執行用戶端應用程式需要使用的應用程式伺服器的機器。當用戶端應用程式初次連結應用程式伺服器時，TSimpleObjectBroker 會從它管理的機器之中選擇一個給 TDCOMConnection 或是 TSocketConnection 元件。TDCOMConnection 或是 TSocketConnection 元件再根據 TSimpleObjectBroker 回傳的機器連結過去並且取得應用程式伺服器。至於 TSimpleObjectBroker 使用什麼方式選擇它管理的機器在下一小節會討論。下圖是這個範例中 TSimpleObjectBroker 管理的機器。

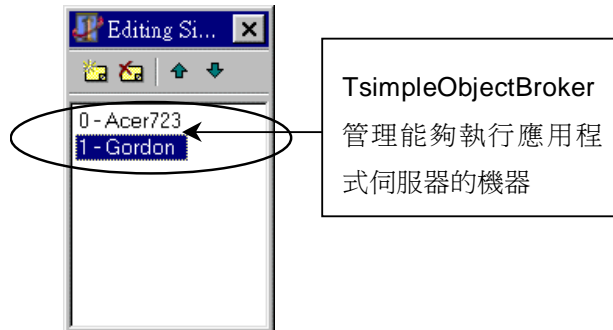


圖 5-12 能夠執行應用程式伺服器的機器

5. 設定完成 TSimpleObjectBroker 元件之後，就可以連結 TSimpleObjectBroker 和 TSocketConnection 在一起了。請點選 TSocketConnection 然後在物件檢視器中設定它的 ObjectBroker 特性值為表格中的 TSimpleObjectBroker 元件。

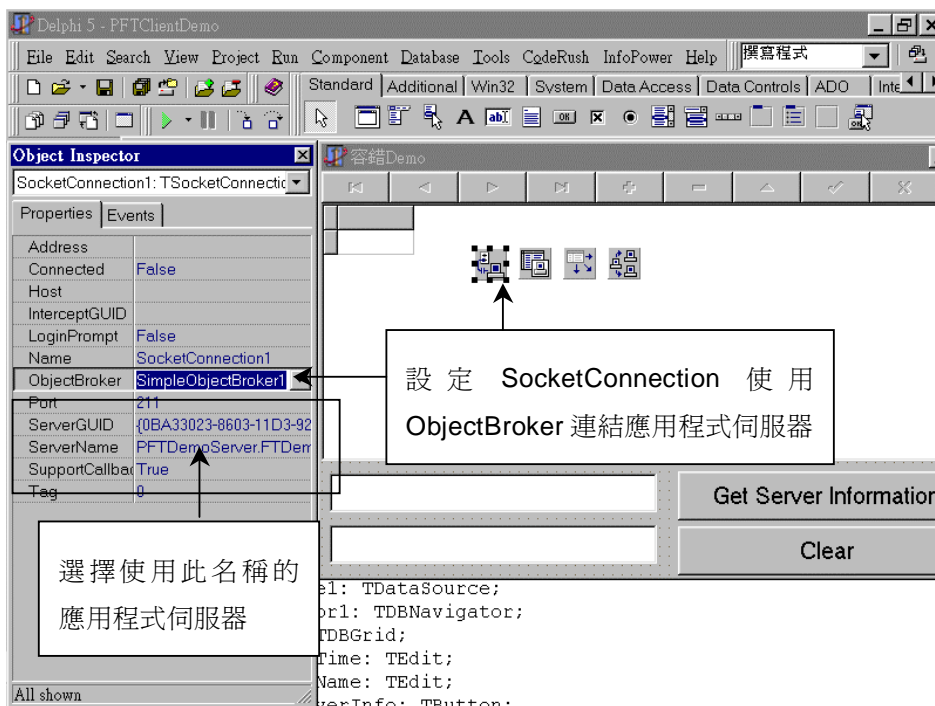
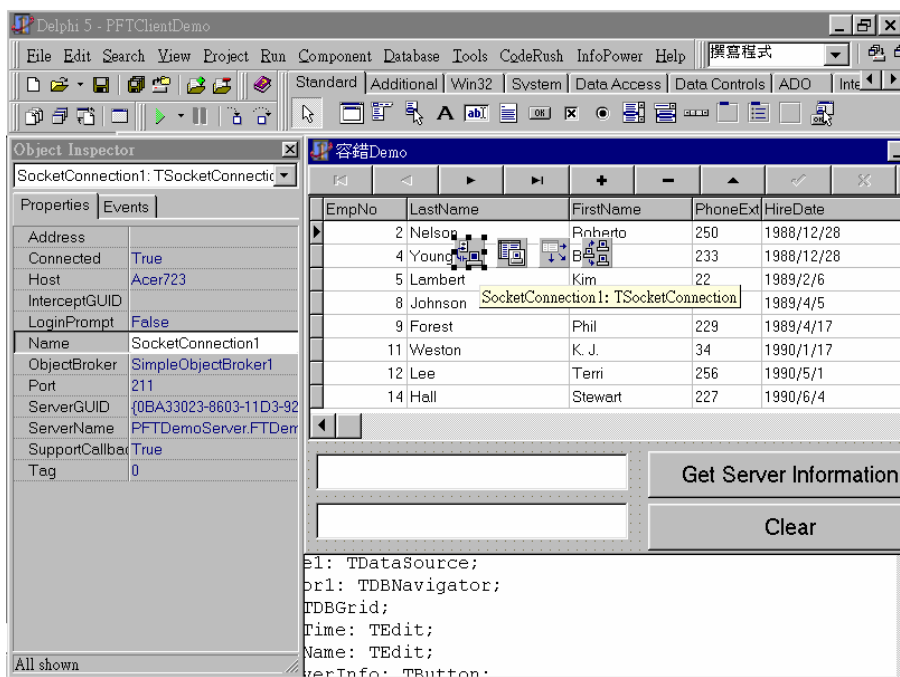


圖 5-13 設定 TSocketConnection 元件的 ObjectBroker 特性值

6. 設定好 TSocketConnection 的 ObjectBroker 特性值之後，就可以開啟 TSocketConnection 連結到遠端的應用程式伺服器了。當程式師設定

TSocketConnection 的 Active 為 True 之後，TSimpleObjectBroker 元件會自動選擇一個它管理的機器，然後把這個機器傳給 TSocketConnection，TSocketConnection 會把這個機器填入它的 Host 特性值中，並且使用這個機器連結應用程式伺服器。例如下圖設定了 TSocketConnection 的 Active 為 True 之後，TSimpleObjectBroker 便會選擇從 Acer723 機器連結應用程式伺服器。



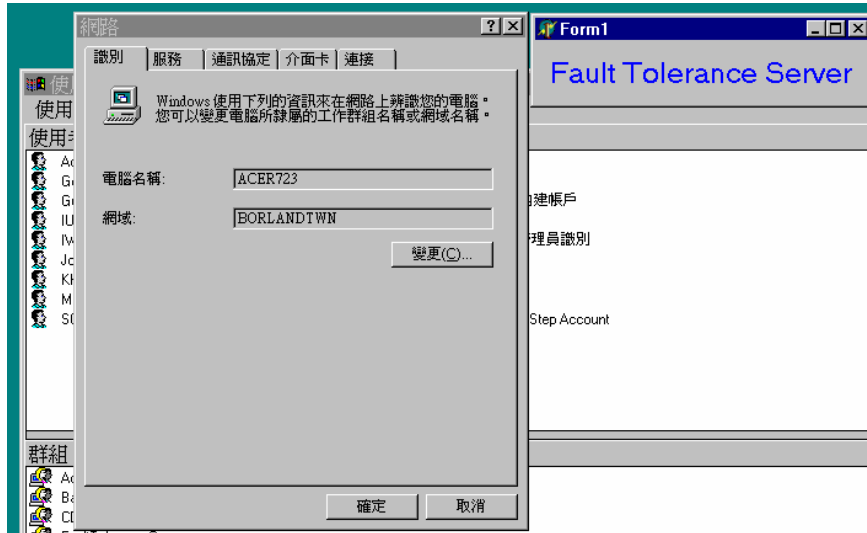


圖 5-14 TSimpleObjectBroker 自動設定 TSocketConnection 使用的機器

上圖有兩個畫面，其中 Delphi 整合發展環境是在 Gordon 的機器中，而 Fault Tolerance Server 則是在 Acer723 機器之中執行。

設定好用戶端應用程式之中的元件之後，程式師必須撰寫一些程式碼以實作出容錯能力的應用程式伺服器。容錯能力實作的觀念很簡單，當用戶端應用程式連結到應用程式伺服器之後，它就可以向應用程式伺服器要求服務。但是當應用程式伺服器故障時，用戶端應用程式如果向應用程式伺服器要求服務，那麼用戶端應用程式就會產生一個錯誤例外。這個時候用戶端可以呼叫 TSimpleObjectBroker 的 SetConnectStatus 方法以設定目前的應用程式伺服器成為不堪使用的狀態，然後再呼叫 TSimpleObjectBroker 的 GetComputerForProgID 方法向 TSimpleObjectBroker 要求另外一個可以使用的機器，以便在這個新的機器中連結提供相同服務的應用程式伺服器。最後再呼叫新取得的應用程式伺服器以取得服務。這個過程可以使用下面的圖形來說明。

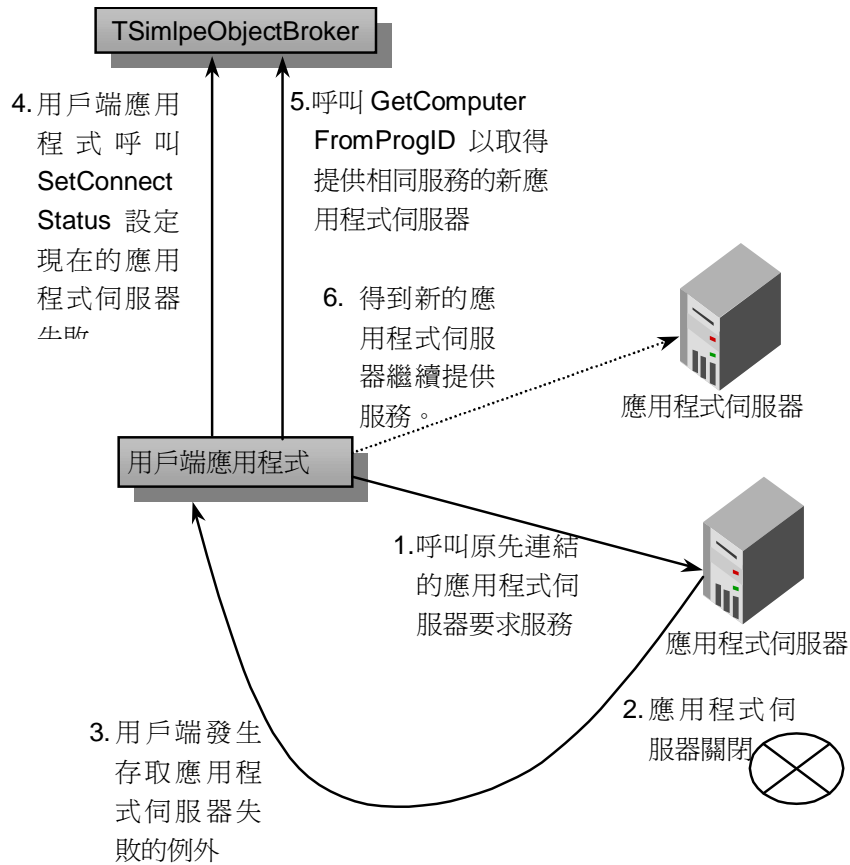


圖 5-15 實作容錯能力的流程

瞭解了如何實作容錯能力的觀念之後，就可以開始撰寫程式碼了。下面就是實作上圖容錯流程的程式碼。

```

procedure TForm2.GetNewServer;
begin
    SocketConnection1.Connected := False;
    SocketConnection1.Host := SimpleObjectBroker1.GetComputerForProgID
    ('PFTDemoServer.FTDemoServer');
    SocketConnection1.Connected := True;
end;

procedure TForm2.btnGetServerInfoClick(Sender: TObject);
var
    vServerTime : OleVariant;
    vServerName : OleVariant;
    
```

```
begin
  try
    SocketConnection1.AppServer.GetSystemTime(vServerTime);
    SocketConnection1.AppServer.GetServerName(vServerName);
    edtSystemTime.Text := vServerTime;
    edtServerName.Text := vServerName;
  except
    on Exception do
      begin
        SimpleObjectBroker1.SetConnectStatus (SocketConnection1.Host,
False) ;
        GetNewServer;
        btnGetServerInfoClick(Sender) ;
      end;
    end;
  end;

procedure TForm2.FormCreate(Sender: TObject);
begin
  SocketConnection1.Connected := True;
  ClientDataSet1.Active := True;
end;

procedure TForm2.FormDestroy(Sender: TObject);
begin
  ClientDataSet1.Active := False;
  SocketConnection1.Connected := False;
end;

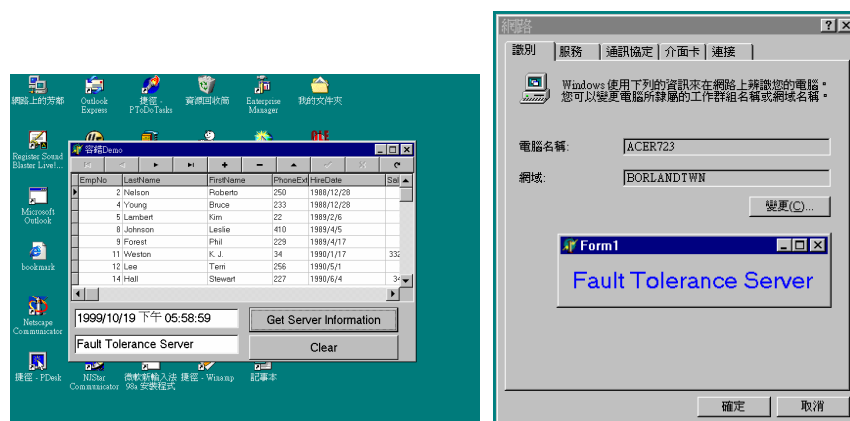
procedure TForm2.btnClearClick(Sender: TObject);
begin
  edtSystemTime.Clear;
  edtServerName.Clear;
end;
```

在上面的程式碼中，用戶端應用程式首先在表格的 OnCreate 事件處理函式中啟動 TSocketConnection，此時 TSocketConnection 就會向 TSimpleObjectBroker 要求一個可以連結的機器，然後 TSocketConnection 就在這台機器之中連結它需要的應用程式伺服器。接著 TClientDataSet 就會開啟從應用程式伺服器中取得資料。

Button1Click 是當使用者按下表格中的按鈕向應用程式伺服器要求服務時執行的方法。Button1Click 首先向應用程式伺服器取得 ServerTime 和 ServerName 特性值，此時如果先關閉它連結的應用程式伺服器，那麼用戶端應用程式就會產生一個例外。當例外產生時用戶端應用程式首先呼叫 TSimpleObjectBroker 的 SetConnectStatus 設定目前的應用程式伺服器成為不可使用的狀態，然後呼叫 GetNewServer 以取得一個新的應用程式伺服器，最後再呼叫 Button1Click 一次。

GetNewServer 首先關閉 TSocketConnection，然後呼叫 TSimpleObjectBroker 的 GetComputerForProgID 以便從 TSimpleObjectBroker 中取得一個可以提供相同應用程式伺服器的機器。TSocketConnection 再使用這個機器連結應用程式伺服器。

在下面的三個圖形中你可以看到用戶端應用程式一開始執行後，便會啟動在 Acer723 機器之中的 Fault Tolerance Server 並且在點選表格中的『Get Server Information』按鈕之後便可以取得伺服器的資訊。現在讓我們強迫關閉 Acer723 機器中的 Fault Tolerance Server，然後再表格中的『Get Server Information』按鈕，那麼就會看到用戶端應用程式便會啟動 Gordon 機器之中的 Fault Tolerance Server，並且再次取得伺服器的資訊。在下圖中第三個畫面便是從 Gordon 機器中取得伺服器時間以及伺服器名稱，請注意第三個圖形和第一個圖形的時間差異。



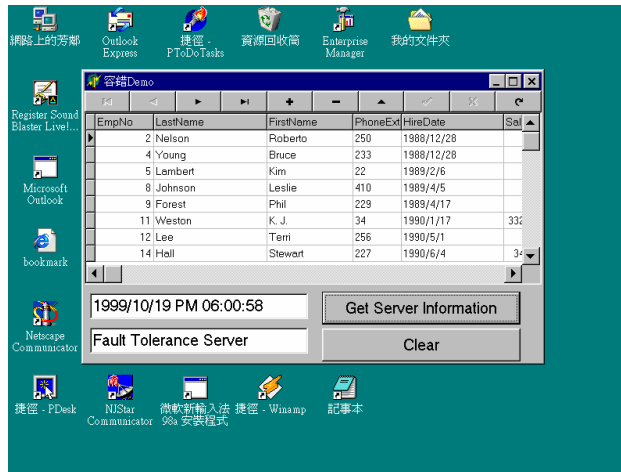


圖 5-16 範例用戶端應用程式執行的畫面

請注意在這個範例中我們之所以能夠使用 TSimpleObjectBroker 元件實作容錯能力是因為範例用戶端應用程式只呼叫應用程式伺服器中的 GetServerName 以及 GetServerTime 這兩個方法。而這兩個方法由於沒有維護任何的狀態資訊，因此這個時候應用程式伺服器可以說是一個無狀態物件，所以用戶端應用程式才能夠使用容錯能力在一台應用程式伺服器故障時再動態的連結另外一台應用程式伺服器要求服務。如果在這個範例中用戶端應用程式是呼叫應用程式伺服器取得資料的話，那麼由於應用程式伺服器不是一個無狀態物件，所以當一台應用程式伺服器故障時，用戶端應用程式並不可以動態連結到另外一台應用程式伺服器再要求資料，因為此時新的應用程式伺服器並不知道用戶端應用程式需要的下一個資料封包是那一個。那麼如果你真的希望在存取資料時也能夠具備容錯能力的話，那麼應該怎麼辦呢？

很簡單，你只需要結合 5-1 討論的『魚與熊掌兼得之法』和 TSimpleObjectBroker 便可以在多台機器中真正的提供容錯的能力。你可以看到 Delphi 的威力，藉由組合數個技術，應用系統可以提供令人不可置信的先進功能。

5-3 開發具備平均負荷能力的應用系統

平均負荷的觀念是指當有數個能夠相同應用程式伺服器的機器時，當有許多用戶端應用程式需要連結應用程式伺服器，MIDAS 能夠分配不同的用戶端應用程式到每一個機器之中，以便平均每一個應用程式伺服器的負荷。Delphi 5 的 TSimpleObjectBroker 也提供平均負荷的功能。要讓應用程式伺服器提供平均負荷的功能，程式師只需要設定它的 LoadBalanced 特性值為 True 就可以提供簡單的平均負荷能力。下圖就是設定 TSimpleObjectBroker 的 LoadBalanced 特性值為 True 以提供平均負荷的能力。

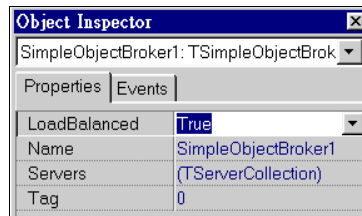


圖 5-17 設定 TSimpleObjectBroker 的 LoadBalanced 特性值

從上面的說明可以知道在 Delphi 5 中應用程式伺服器要提供平均負荷的能力只需要設定一個特性值即可。目前 Delphi 5 是使用隨機的方式在正常執行的機器之中選擇一個機器提供給用戶端。在未來 Delphi 將會提供更精細的平均負荷運算法則。例如根據機器的硬體設備，機器目前連結的用戶端數目，或是機器和用戶端應用程式之間以及和資料庫伺服器之間的距離來決定使用那一個機器。



Microsoft 宣稱在 Windows 2000 之中的 COM+ 將會提供平均負荷的能力，允許用戶端應用程式根據每一台應用程式伺服器的負荷情形來決定繫結到那一台機器之中的應用程式伺服器。但是 COM+ 的平均負荷能力只限於用戶端應用程式第一次啟動時才提供平均負荷的能力，一旦用戶端應用程式啟動之後，便一直會使用特定機器之中的應用程式伺服器，而不會在用戶端應用程式執行的過程中持續的提供平均負荷的能力。這和 MIDAS 3.0 提供的平均負荷能力幾乎是一樣的。但是 Microsoft 在 1999 年的 10 月份又表示

COM+ 不一定會提供平均負荷的能力，因此 MIDAS 3.0 是目前唯一提供平均負荷能力的中介技術。

5-4 安全強固的應用系統

在分散式計算環境中，程式師開發的應用系統除了必須能夠正確而且有效率的運作之外，如何讓應用系統更為安全強固，不會因為應用程式伺服器或是資料庫伺服器故障而導致整個應用系統無法繼續運作也是非常重要的。例如設你開發了一個多層分散式應用系統，其中包含了數十，或是數百台的用戶端機器，數台應用程式伺服器，以及一台資料庫伺服器。有一天所有的使用者正在輸入資料，而且每一個人已經在用戶端機器中輸入了數十筆的資料，但是就在使用者按下 ApplyUpdates 按鈕之前，突然有一台應用程式伺服器故障了，那麼所有連結到這台應用程式伺服器的用戶端使用者如何異動他們輸入的資料到資料庫伺服器之中呢？

如果更不幸的是此時是資料庫伺服器故障，或是所有的應用程式伺服器故障了，那麼所有使用者輸入的資料又應該怎麼辦呢（假設現在有 100 個使用者，而且每一個使用者輸入了 15 筆資料，那麼就可能流失 1500 筆的資料）？以前我曾見過這種情形，通常的結果是使用者必須關閉應用程式，等待故障的機器修復好，最後重新啟動應用程式伺服器，執行用戶端應用程式，重新輸入所有的資料（並且祈禱這種情形不會再發生），再點選 ApplyUpdates 按鈕更新資料。

在分散式計算環境中這種情形發生的可能性也許會更高，因為程式師如果使用 COM/DCOM 的技術，那麼應用程式伺服器和資料庫伺服器都可能執行在 Windows 平台之中，如何為多層分散式應用系統內建合理的容錯能力是程式師必須思考的問題。

在前面的小節中已經討論了具備簡單容錯能力的應用系統，本小節將會對於如何開發安全強固的多層分散式應用系統做深入的討論。這些討論的內容都是我在實作多層應用系統時累積的經驗，我認為這些容錯能力的實際技術是非常重要的，因為你永遠無法預測當你的多層應用系統分發出去執行時會發生什麼狀況。在我撰寫本章時，我曾試著搜尋任何在 Windows 平台中實作多層應用系統的書籍，不管這些書籍討論的開發工具是 Delphi，Visual Basic，或是

PowerBuilder，幾乎都沒有任何的書籍討論如何實作安全強固的多層應用系統。現在不管是 Inprise 或是 Microsoft 都強調多層應用系統的重要性，但是為什麼沒有作者撰寫任何有關如何實作安全強固多層應用系統的內容呢？我相信本小節的內容和技術對於任何真的要開發多層應用系統的人都是非常重要的。

在我們開始學習如何實作安全強固的多層應用系統之前，瞭解一些基本的觀念以及知道現在 Delphi 提供的基本容錯能力是非常重要的。因為這樣可以讓程式師知道開發工具的限制，以及程式師開發的多層應用系統如何能夠克服這些限制進而實作出更好的容錯能力。

讓我們從一個簡單的多層應用系統執行的情形開始說明。圖 5-18 是一個假設的多層應用系統架構圖。在這個圖形中有兩個應用程式伺服器，另外還有一個用戶端應用程式，此外在圖 5-18 中我把 TSimpleObjectBroker 使用一個額外的圖像來顯示以方便說明之用。

假設現在用戶端應用程式已經異動了資料，並且呼叫 ApplyUpdates 把異動的 Delta 封包傳遞給應用程式伺服器 1，要求應用程式伺服器 1 更新資料庫中的資料。

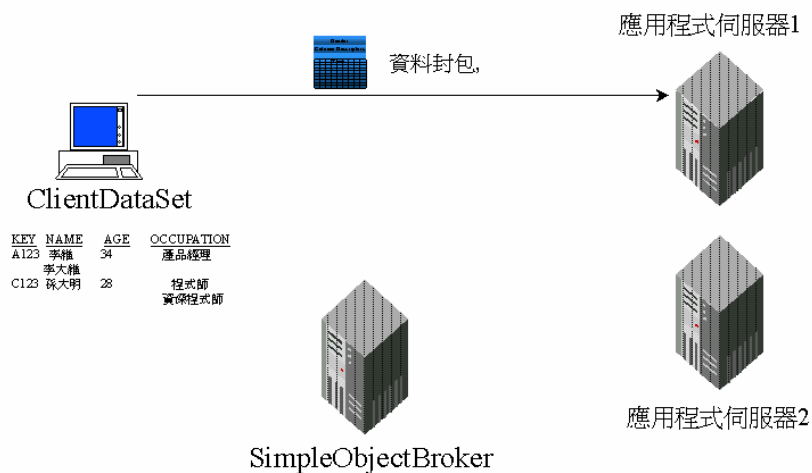


圖 5-18 範例多層應用系統架構圖

如果在用戶端應用程式呼叫 `ApplyUpdates` 異動資料的那一剎那應用程式伺服器 1 突然故障了。這會導致 `ApplyUpdates` 執行失敗，而且作業系統的 COM/DCOM 執行時期函式館會回傳一個例外錯誤。當圖 5-19 的情形發生時，用戶端應用程式必須能夠處理這個例外錯誤，否則用戶端應用程式異動的資料可能會流失。如果此時有另外一個應用程式伺服器啟動並且向 `TSimpleObjectBroker` 註冊，或者是在 `TSimpleObjectBroker` 中已經有其他的應用程式伺服器存在的話，那麼用戶端應用程式就可以使用這些額外的應用程式伺服器來處理異動的資料。問題是程式師知道怎麼讓 `TSimpleObjectBroker` 提供另外一個提供相同服務的應用程式伺服器給用戶端應用程式呢？（`TSimpleObjectBroker` 果然夠 Simple，和 CORBA 的 `OsAgent` 比起來是遜色了許多）

現在假設在區域網路中有另外一台機器執行了應用程式伺服器 2。圖 5-20 敘述了這個情形，應用程式伺服器 2 向 `TSimpleObjectBroker` 註冊。

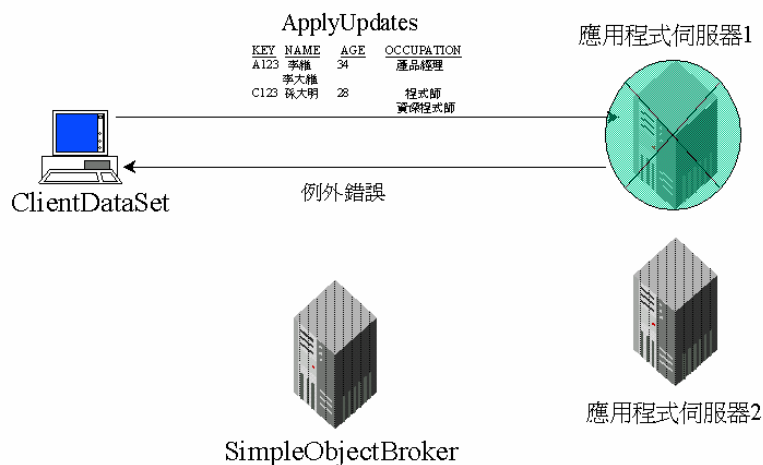


圖 5-19 應用程式伺服器 1 故障導致錯誤發生

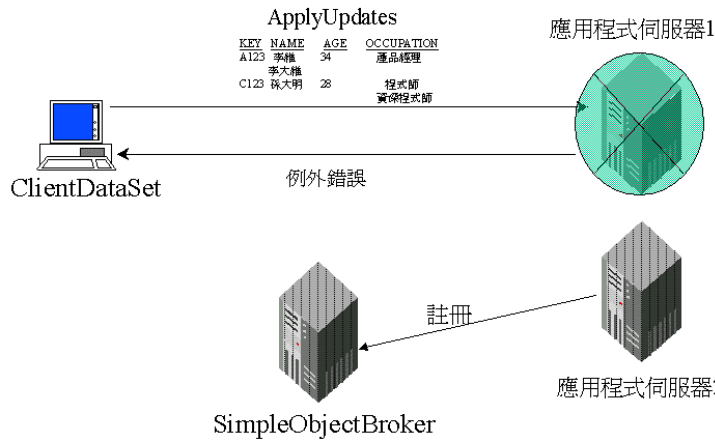


圖 5-20 應用程式伺服器 2 啟動並且向 TSimpleObjectBroker 註冊

當圖 5-20 的情形發生後，用戶端應用程式可以遵照圖 5-21 的步驟和 TSimpleObjectBroker 互動。首先當用戶端應用程式查覺應用程式伺服器 1 已經故障後，可以立刻呼叫 TSimpleObjectBroker 的 SetConnectedStatus (False) 通知 TSimpleObjectBroker 應用程式伺服器 1 發生了故障無法繼續使用，然後呼叫 GetComputerForProgID 要求 TSimpleObjectBroker 搜尋另外一台提供相同服務的應用程式伺服器給用戶端應用程式。

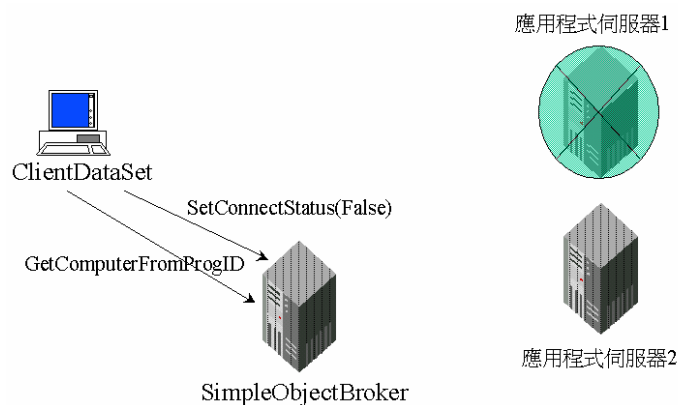


圖 5-21 用戶端應用程式通知 TSimpleObjectBroker 應用程式伺服器 1 故障，並且要求 TSimpleObjectBroker 回傳一個提供相同服務的應用程式伺服器讓用戶端應用程式使用

當 TSimpleObjectBroker 接受到 GetComputerForProgID 的呼叫後，它會在它維護的應用程式伺服器目錄中搜尋一台提供相同服務而且可以工作的應用程式伺服器回傳給用戶端應用程式。

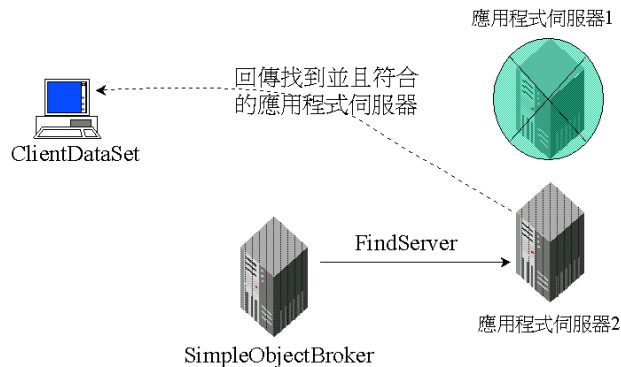


圖 5-22 TSimpleObjectBroker 找到另外一個提供相同服務的應用程式伺服器並且回傳給用戶端應用程式呼叫

當用戶端應用程式再次從 TSimpleObjectBroker 收到 TSimpleObjectBroker 提供的應用程式伺服器後，就可以呼叫這個新的應用程式伺服器的 ApplyUpdates 方法要求它異動資料。

請注意，在這裡有一個非常重要的觀念，那就是 Delphi 5 的 MIDAS 3.0 對於用戶端應用程式異動資料 (Delta) 的實作技術。在第 4 章已經說明過用戶端應用程式對於應用程式伺服器傳遞到用戶端資料所有的異動都是儲存在 TClientDataSet 的 Delta 特性值之中。當用戶端應用程式呼叫 TClientDataSet 的 ApplyUpdates 方法更新異動資料時，是把用戶端的 Delta 特性值傳遞給應用程式伺服器中 TDataSetProvider 元件的 ApplyUpdates 方法。所以一個用戶端應用程式維護的 Delta 特性值是一個很奇特的特性，它的值 (即異動的資料) 是獨立於任何應用程式伺服器之外的。這個意思是說一個用戶端應用程式的 Delta 值可以傳遞給任何提供相同服務的應用程式伺服器更新資料，只要這個應用程式伺服器和原先的應用程式伺服器都提供相同服務，而且都有一個 TDataSetProvider 元件是連結到要更新的資料表就可以了。

如果你還不瞭解這段話的意思，那麼讓我們使用一個例子來解釋。就以圖 5-18 開始的範例來說明，如果用戶端應用程式當初是從應用程式伺服器 1 中

藉由命為 pdrPersion 的 TDataSetProvider 元件所連結的 Persons 的資料表中取得圖 5-18 李維，孫大明等人的資料。那麼當用戶端應用程式異動了這些資料之後，用戶端應用程式的 TClientDataSet 就包含了這些異動資料。如果現在用戶端應用程式呼叫 ApplyUpdates 更新資料時應用程式伺服器 1 故障了，只要用戶端應用程式能夠再找到應用程式伺服器 2，而且應用程式伺服器 2 也包含了 Persons 資料表以及輸出一個連結到 Persons 資料表的 TDataSetProvider 元件的話，那麼用戶端應用程式就可以呼叫應用程式伺服器 2 中連結到 Persons 資料表的 TDataSetProvider 元件的 ApplyUpdates 方法更新資料。

許多人可能無法相信這樣的過程，因為應用程式伺服器 1 和用戶端應用程式之間會維護一些內部的資訊，用戶端應用程式如何能夠讓應用程式伺服器 2 更新資料呢？這是因為在用戶端應用程式和應用程式伺服器 1 中 Delphi 會維護一個 cursor 的資訊，這個資訊只是為了讓應用程式伺服器能夠知道用戶端應用程式下一次需要的資料封包是那一個。但是這個 cursor 資訊和異動資料是沒有什麼關係的，TDataSetProvider 完全是根據用戶端應用程式傳遞來的 Delta 做為更新資料的依據。所以即使應用程式伺服器 1 故障了，只要用戶端應用程式仍然保有 Delta 特性值，那麼用戶端應用程式就有機會找到其他的應用程式伺服器為它更新資料。更何況在 MIDAS 3.0 中程式師可以建立無狀態物件，在這種情形下 TDataSetProvider 根本不會為用戶端應用程式維護的 cursor 資訊。事實上現在 MIDAS 3.0 的無狀態物件不論是在用戶端應用程式存取資料，以及更新資料的行為上都使用了一致的觀念和流程，那就是都使用無狀態的觀念來處理資料。因此在整個處理資料的模式也就更為一致了，這樣是一個比較好的模式，也是未來 MIDAS 更應該發展的方向。

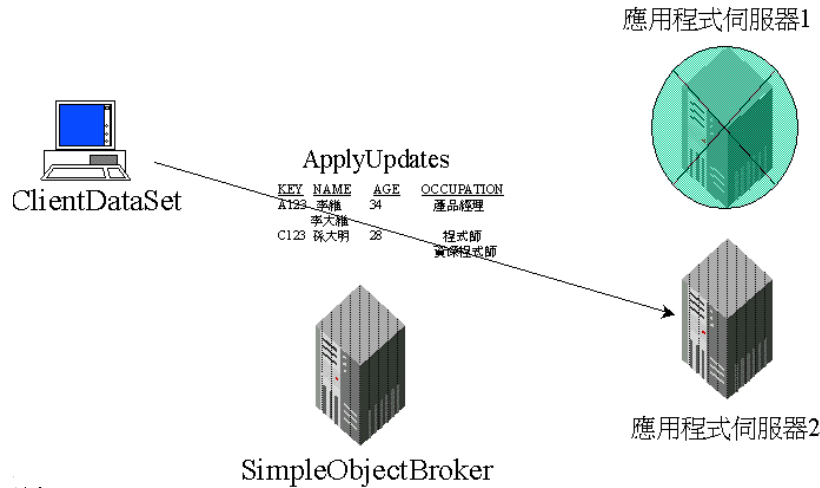


圖 5-23 用戶端應用程式呼叫新取得的應用程式伺服器以更新異動的資料

圖 5-24 即說明了用戶端應用程式和應用程式伺服器對於資料維護的關係。從圖中可以知道即使用戶端應用程式是使用應用程式伺服器 2 真正的更新資料，它只是失去了 cursor 的資訊，Delta 中異動的資料仍然能夠更新回資料表之中，事實上我們甚至可以在應用程式伺服器 2 中回復應用程式伺服器 1 中維護的 cursor 的資訊。

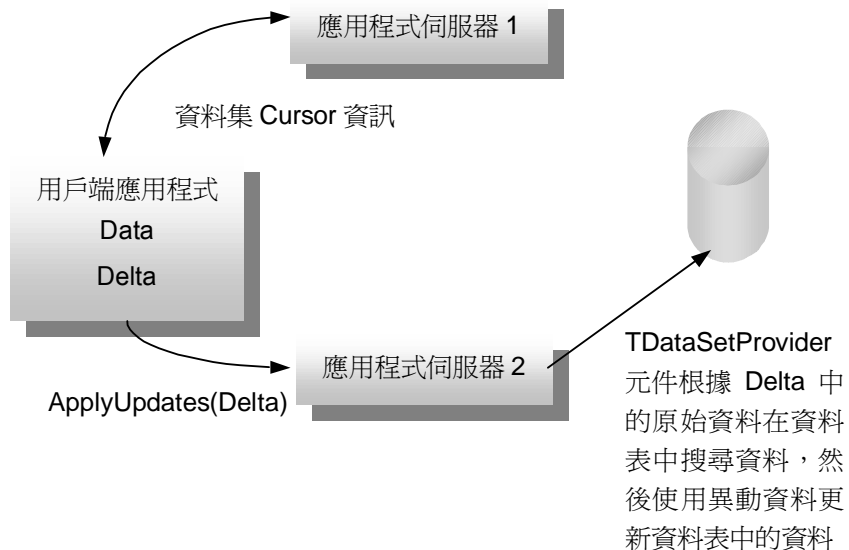


圖 5-24 用戶端應用程式和應用程式伺服器中 Provider 元件的互動

瞭解了用戶端應用程式和應用程式伺服器之間對於異動資料的維護原理後，程式師就可以實作出非常安全強固的容錯多層應用系統，在稍後實作的章節中會看到實際運作的程式碼。現在再讓我們繼續範例的討論。

圖 5-25 顯示的是稍後應用程式伺服器 1 又回復了功能，這時它可以通知 TSimpleObjectBroker 把它的狀態回復成可提供服務的應用程式伺服器。

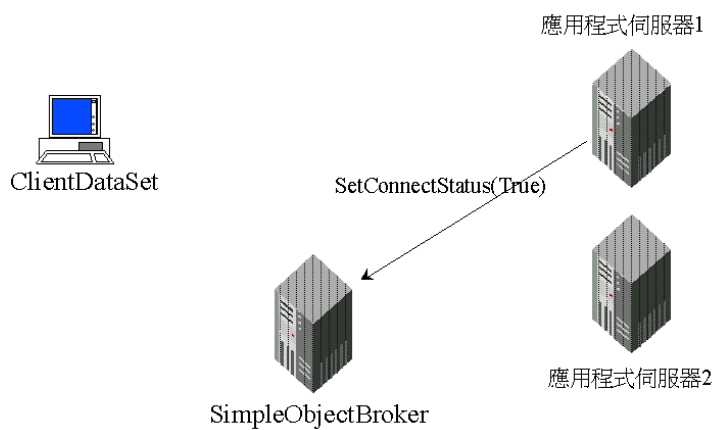


圖 5-25 原先故障的應用程式伺服器 1 回復功能並且通知 TSimpleObjectBroker

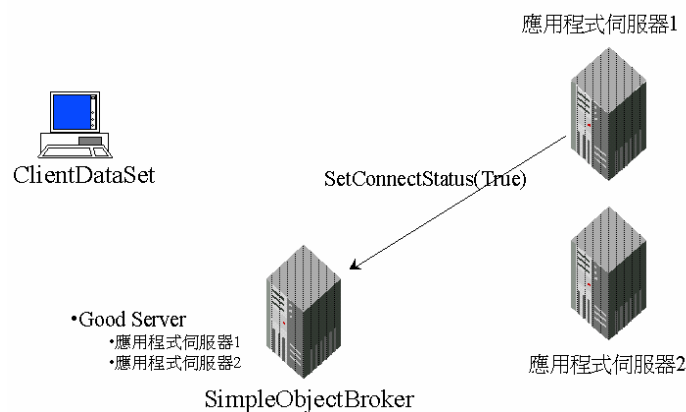


圖 5-26 TSimpleObjectBroker 維護的應用程式伺服器目錄重新登錄可正常工作的應用程式伺服器

當 TSimpleObjectBroker 知道了應用程式伺服器 1 又可以提供服務後，那麼 TSimpleObjectBroker 內部維護的應用程式伺服器目錄就回復成最原始的狀

態，此時 TSimpleObjectBroker 內部的應用程式伺服器目錄中能夠提供服務的應用程式伺服器又有了應用程式伺服器 1 和應用程式伺服器 2。



圖 5-26 中 SimpleObjectBroker 維護的應用程式伺服器目錄這種觀念和技術在 CORBA 中早已實作出來。例如上一章中討論的 OsAgent 也會維護類似的目錄。在未來 COM+ 中，Microsoft 也會實作類似的應用程式伺服器目錄，提供動態的應用程式伺服器狀態的資訊。

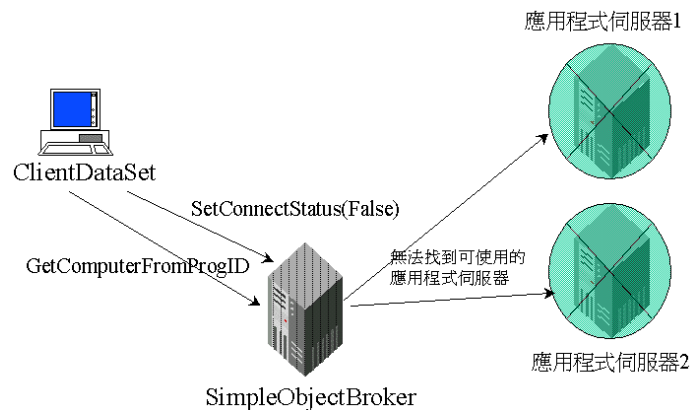


圖 5-27 TSimpleObjectBroker 維護的應用程式伺服器 2 也故障了，用戶端應用程式再次呼叫 GetComputerForProgID 企圖取得另外一台應用程式伺服器

現在再讓我們假設圖 5-27 中最壞的狀況出現了，那就是當用戶端應用程式欲藉由應用程式伺服器 2 更新資料時，應用程式伺服器 2 也故障了，所以用戶端應用程式只好再執行一次 SetConnectedStatus 和 GetComputerForProgID 方法要求 TSimpleObjectBroker 再為它搜尋另外一個能夠提供相同服務的應用程式伺服器。但是此時所有的應用程式伺服器都已經故障，所以 TSimpleObjectBroker 無法再找到任何能夠提供服務的應用程式伺服器，那麼用戶端應用程式異動的資料要怎麼辦呢？這個時候是不是使用者只好關閉用戶端應用程式，等到應用程式伺服器回復運作時再重新輸入資料呢？

當然如果你的多層應用系統是這樣運作而且使用者也接受的話，那麼也許沒有什麼問題。但是你可以提供更好的容錯能力讓使用者在所有應用程式伺服

器故障時仍然能夠儲存已經異動的資料，等到應用程式伺服器回復時再自動的更新這些異動的資料。圖 5-28 就顯示了這個技術。

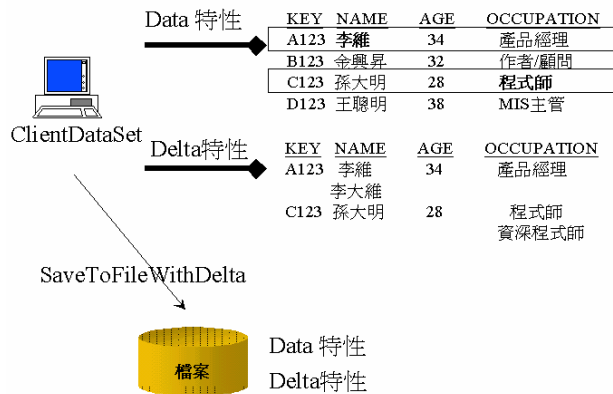


圖 5-28 TSimpleObjectBroker 無法提供額外的應用程式伺服器，所以用戶端應用程式只好先暫時的把原始資料 (Data) 以及異動資料 (Delta) 儲存在用戶端的檔案之中

在圖 5-28 中當用戶端應用程式無法藉由任何的應用程式伺服器更新資料時，它可以試著把異動的資料儲存在本地機器的硬式磁碟機之中，等到應用程式伺服器回復運作時，再載入這些異動資料然後更新回資料表之中。TClientDataSet 提供了一個 SaveToFile 的方法，但是很不幸的是 TClientDataSet 的 SaveToFile 在儲存用戶端應用程式異動的資料時有一些問題的存在，在稍後會詳細的說明。在圖 5-29 中顯示程式碼呼叫 SaveToFileWithDelta 這個方法同時儲存 TClientDataSet 的 Data 和 Delta 特性值。在稍後實作的章節中會討論這個方法的功能。

當 TClientDataSet 儲存了 Data 和 Delta 之後，那麼當應用程式伺服器回復功能時，它就可以像圖 5-29 所示呼叫 LoadFromFileWithDelta 從硬式磁碟機中載入原先儲存的 Data 和 Delta 回到 TClientDataSet 之中，然後像圖 5-30 一樣呼叫 GetComputerForProgID 取得回復功能的應用程式伺服器，最後再如圖 5-31 顯示的呼叫取得的應用程式伺服器的 ApplyUpdates 方法更新資料。

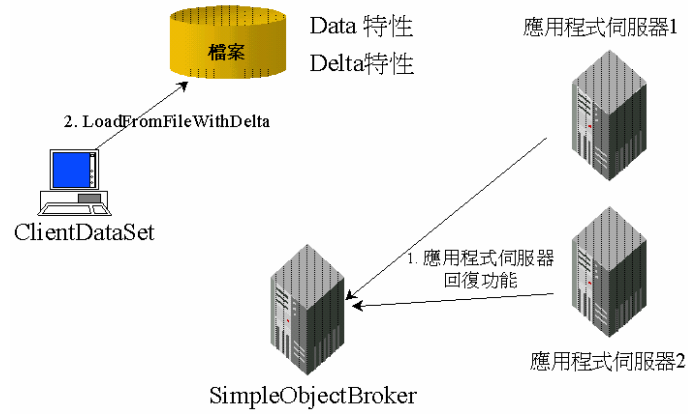


圖 5-29 稍後，所有的應用程式伺服器都回復了功能並且通知 TSimpleObjectBroker，而用戶端應用程式也把先前暫時儲存的原始資料以及異動資料載入回 TClientDataSet 之中

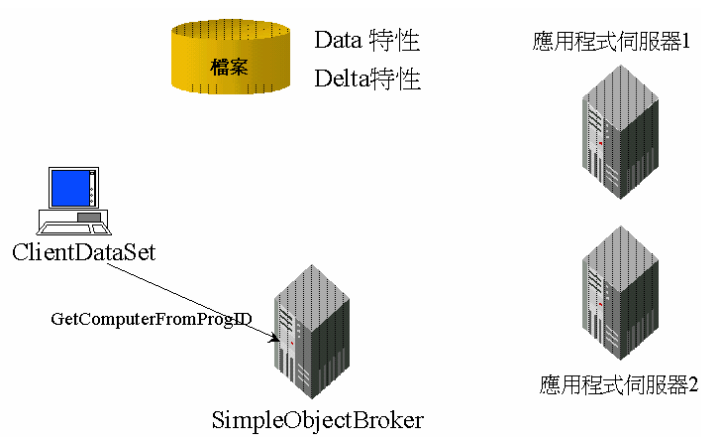


圖 5-30 用戶端應用程式可以再次呼叫 GetComputerForProgID 取得回復的應用程式伺服器

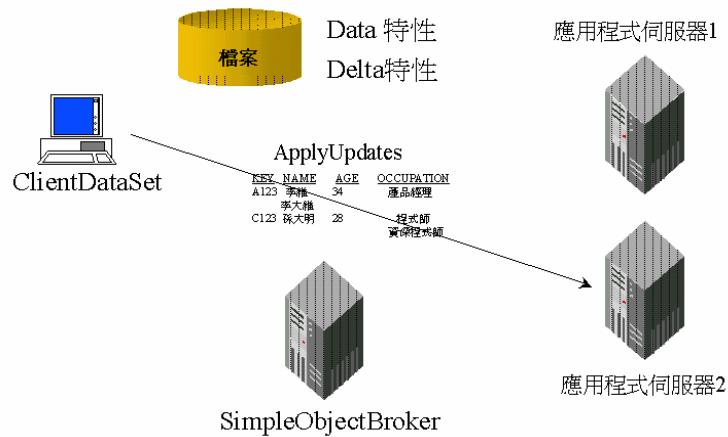


圖 5-31 最後用戶端應用程式再呼叫應用程式伺服器的 ApplyUpdates 異動資料回資料庫

從前面的討論中可以各位可以知道如何使用 Delphi 5 開發一個安全強固的多層應用系統。但是光瞭解這些觀念是不夠的，程式師需要實際的開發出這種安全強固的多層應用系統。在下面的小節中本書將會以數個範例程式告訴各位如何把這些觀念轉換為真正的程式碼。

這些範例除了展示如何開發安全強固的分散式多層應用系統之外，也蘊含了許多重要的程式技巧。瞭解並且學習這些程式技巧可以讓程式師更靈活的運用 Delphi 5 撰寫出更為精巧的應用系統，此外在最後一個小節中你將會看到我們如何運用對於 MIDAS 的深入瞭解以克服目前 Delphi 的限制，進而實作出理想的多層應用系統。

5-5 開發使用 COM/DCOM 技術的安全強固應用系統

本小節將使用 COM/DCOM 的技術實作出一個能夠在應用程式伺服器故障時，用戶端應用程式仍然能夠藉由另外一台提供相同服務的應用程式伺服器來完成資料的異動。如此一來使用者在用戶端應用程式中輸入的資料便不會因為它一開始連結的應用程式伺服器失敗時造成資料的流失。

在這個範例應用系統中，我們先建立一個應用程式伺服器，這個應用程式伺服器只是簡單的更新本書範例資料庫中的 SAMPLETBL 資料表：

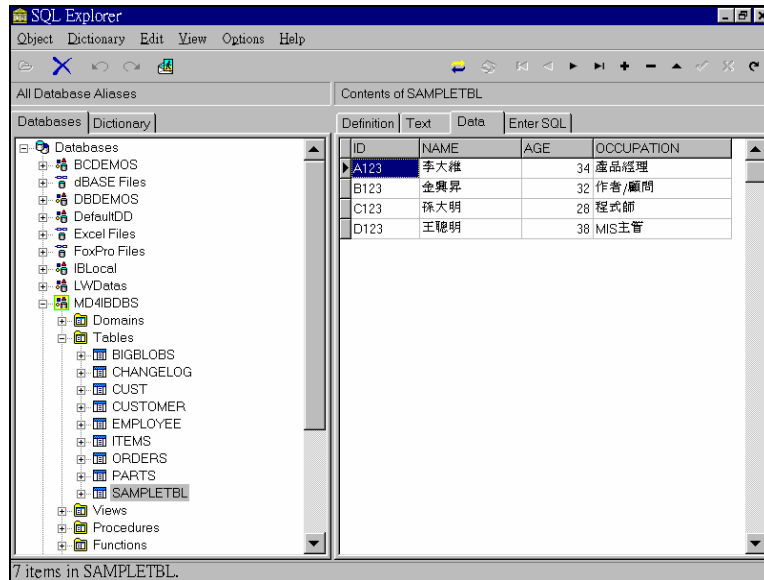


圖 5-32 SAMPLETBL 範例資料表

用戶端應用程式藉由 TSimpleObjectBroker 連結到此應用程式伺服器異動資料。首先讓我們先撰寫應用程式伺服器。

5-4-1 撰寫應用程式伺服器

這個應用程式伺服器建立了一個遠端資料模組，並且在遠端資料模組中置入 TSession，TDatabase，TQuery 以及 TDataSetProvider 元件。然後使用 TQuery 元件連結到 SAMPLETBL 資料表。而應用程式伺服器的主表格則使用了 TClientDataSet 元件以及 TDBGrid 以便顯示用戶端傳遞過來的異動資料。

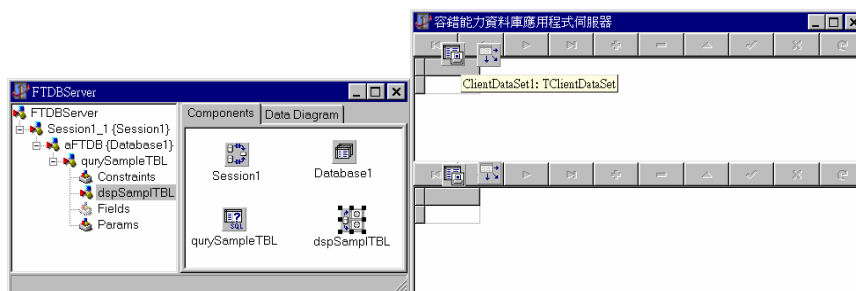


圖 5-33 應用程式伺服器的遠端資料模組以及主表格

當用戶端應用程式呼叫 `ApplyUpdates` 方法異動資料時，範例程式在 `TDataSetProvider` 元件的 `OnUpdateData` 事件處理函式中於應用程式伺服器的主表格中顯示異動資料。這個技巧在前面的章節中已經說明過了，只需要把 `OnUpdateData` 的 `DataSet` 參數的 `Data` 特性值指定給主表格中空白的 `TClientDataSet` 的 `Data` 特性值即可。

```
procedure TMIDAS2Server.pdrMIDASUpdateData(Sender: TObject;  
    DataSet: TClientDataSet);  
begin  
    frmMIDASServerMain.ClientDataSet1.Data := DataSet.Data;  
end;
```

現在這個應用程式伺服器就算是完成了，這個應用程式伺服器和一般的應用程式伺服器似乎沒有什麼不同。沒錯，本節要開發的安全強固多層應用系統主要的機制是在用戶端的應用程式之中。接下來就是這個容錯應用系統的關鍵地方了。

5-4-2 撰寫容錯用戶端應用程式

要撰寫具備容錯能力的用戶端應用程式在觀念上並不困難，只要程式師對於 `TClientDataSet` 的 `Data` 和 `Delta` 特性的運作原理瞭解，那麼撰寫實作程式碼就很簡單。在『深入瞭解 MIDAS』一章中我們已經詳細的討論了 `TClientDataSet` 的 `Data` 和 `Delta` 特性，以及如何熟練的使用它們。所以在開發容錯能力的用戶端應用程式伺服器時，只需要如下的觀念：

當用戶端應用程式開始執行時，它會連結一個應用程式伺服器，並且使用這個連結的應用程式伺服器異動資料。如果在用戶端應用程式異動資料的時候應用程式伺服器因為某些原因故障時，用戶端應用程式只要能夠找到另外一台提供相同服務的應用程式伺服器，再次連結到這台新的應用程式伺服器，然後再次呼叫 `TClientDataSet` 的 `ApplyUpdates` 方法即可。因為應用程式伺服器在異動用戶端應用程式傳遞來的資料時，並不需要維護一個 `context` 的內容。

這個觀念在前面的章節已經介紹過了，如果你還不瞭解的話，建議您先閱讀『深入瞭解 MIDAS』章節。現在我們就可以開始實際的撰寫程式碼了。

首先建立一個新的應用程式，然後在主表格中置入如圖 5-33 的元件，這個用戶端範例程式使用 TSocketConnection 元件來連結剛才建立的應用程式伺服器。當置入了 TSimpleObjectBroker 之後必須向它註冊才可以執行剛才開發的應用程式伺服器的機器。

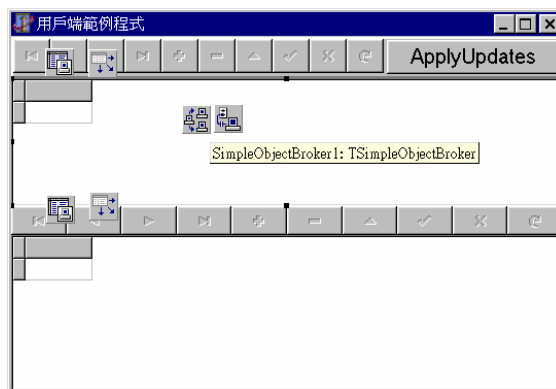


圖 5-34 用戶端應用程式的主表格

請使用滑鼠點選表格中的 TSimpleObjectBroker 元件，然後在物件檢視器中點選它的 Servers 特性，啟動 Servers 特性值編輯器。Delphi 會顯示圖 5-35 的對話盒，請點選對話盒上方的 Add New 按鈕以加入可以執行應用程式伺服器的機器。

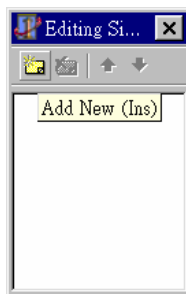


圖 5-35 Servers 特性值編輯器

點選了 Add New 按鈕之後，Delphi 會顯示下圖的瀏覽電腦對話盒，請從這個對話盒中選擇可以執行應用程式伺服器的電腦。例如在這個範例中，Gordon 和 Acer723 都是可以執行上一小節開發的應用程式伺服器，所以我們就如圖 5-37 一樣加入 Gordon 和 Acer723 這個兩台機器的名稱到 Servers 特性

之中。最後再設定 TSimpleObjectBroker 的 LoadBalanced 特性值為 True，以便使用 TSimpleObjectBroker 的平均負荷能力。

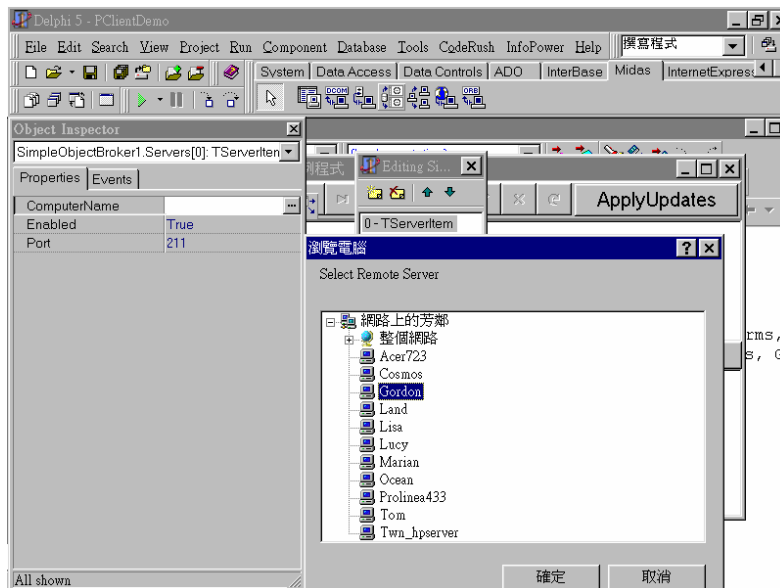


圖 5-36 瀏覽電腦對話盒



圖 5-37 可以執行應用程式伺服器的電腦

在設定好了 TSimpleObjectBroker 的特性值之後，就可以設定主表格中 TSocketConnection 元件的特性值了。請設定 TSocketConnection 的 ObjectBroker 特性值為剛才設定的 TSimpleObjectBroker 元件，然後當你把 TSocketConnection 元件的 Connected 特性值設定為 True 時，TSimpleObjectBroker 便會在這兩台機器 (Gordon 和 Acer723) 中根據平均負荷的狀態動態的選擇一台機器執行應用程式伺服器，然後再讓

TSocketConnection 連結到選定機器之中的應用程式伺服器。請注意，由於我設定了 TSimpleObjectBroker 的 LoadBalanced 特性值為 True，所以每次當我設定 TSocketConnection 元件的 Connected 特性值時，有時候 Gordon 機器的應用程式伺服器會執行，而有時是 Acer723 機器的應用程式伺服器執行。

最後請設定主表格之中的 TClientDataSet 以及連結其他的資料感知元件。現在就可以開始撰寫重要的程式碼了。

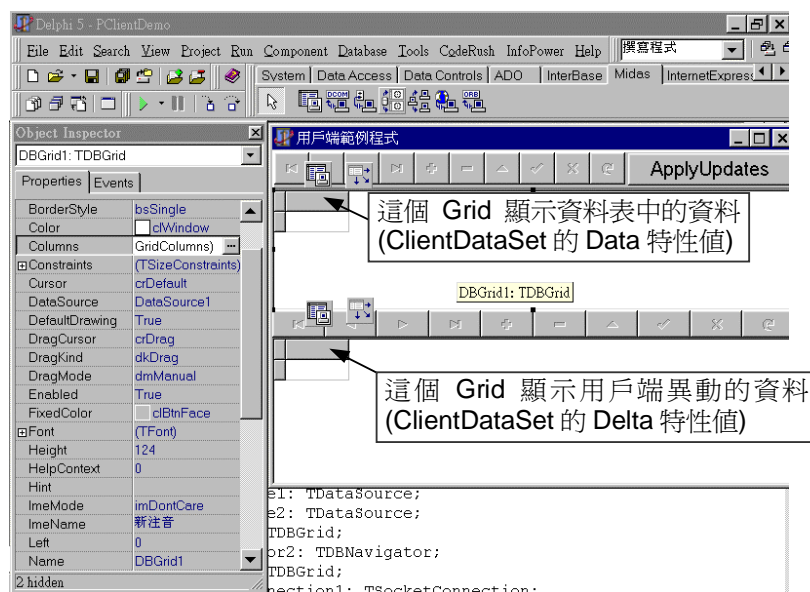


圖 5-38 設定 TSocketConnection 的 ObjectBroker 特性值為 SimpleObjectBroker1

首先我們在主表格一啟動時先顯示它目前連結的應用程式伺服器是那一台機器。

```

procedure TfrmMIDASClientMain.FormActivate(Sender: TObject);
begin
  Self.Caption := 'MIDAS 用戶端應用程式' + ' 現在連結的 Server 是 ' +
    ScketConnection1.Host;
end;

```

然後在 TClientDataSet 的 AfterPost 事件處理函式中把用戶端應用程式異動的資料顯示在主表格下方的 DBGrid 之中。

```

procedure TfrmMIDASClientMain.ClientDataSet1AfterPost(DataSet:
  TDataSet);

```

```
begin
  ClientDataSet2.Data := ClientDataSet1.Delta;
end;
```

最後，當使用者異動完資料並且按下主表格中的 `ApplyUpdates` 按鈕後，用戶端應用程式並不只是簡單的呼叫 `ApplyUpdates` 而已，它使用了下面的程式碼來處理可能發生的問題。

```
procedure TfrmMIDASClientMain.Button1Click (Sender: TObject) ;
var
  sMachine : string;
begin
  try
    ClientDataSet1.ApplyUpdates(0);
  except
    on Exception do
      begin
        try
          SocketConnection1.Connected := False;
          SimpleObjectBroker1.SetConnectStatus
            (SocketConnection1.Host, False);
        finally
          try
            sMachine := SimpleObjectBroker1.GetComputerForProgID
              (SocketConnection1.Host);
            SocketConnection1.Host := sMachine;
            SocketConnection1.Connected := True;
            ClientDataSet1.ApplyUpdates(0);
          except
            on EBrokerException do
              begin
                ShowMessage ('沒有任何可供使用的應用程式伺服器了') ;
                ClientDataSet1.Cancel;
              end;
            end;
          end;
        end;
      end;
  end;
end;
```

在上面的程式碼中，用戶端應用程式首先如平常一樣呼叫 `ApplyUpdates` 方法，但是 `ApplyUpdates` 方法是包含在一個 `try...except` 程式區塊之中。如果 `ApplyUpdates` 發生了例外，那麼就可能是應用程式伺服器故障了。所以在

except 程式區塊中用戶端應用程式先切斷和原先應用程式伺服器的連結，並且呼叫 TSimpleObjectBroker 的 SetConnectStatus 以便通知 TSimpleObjectBroker 原先的應用程式伺服器已經無法繼續使用了。

接著用戶端應用程式又使用了另外一個 try...except 程式區塊，在這個程式區塊中，它先試圖向 TSimpleObjectBroker 取得另外一台能夠提供相同服務的應用程式伺服器，然後重新連結這台新的應用程式伺服器，再呼叫 ApplyUpdates 方法藉由新的應用程式伺服器異動資料回資料表之中。而如果在這個程式區塊中又發生了例外，那麼便是因為 TSimpleObjectBroker 已經沒有其他可以提供相同服務的應用程式伺服器了，所以用戶端應用程式會顯示一個錯誤訊息。上面的程式碼並不困難，只要具備『深入瞭解 MIDAS』一章中的觀念，相信你可以立刻就掌握其中的訣竅。

現在讓我們實際的執行這個應用系統，看看它是否真的能夠在不同的應用程式伺服器之中切換而且繼續的執行。首先拷貝應用程式伺服器到 Gordon 和 Acer723 機器之中，並且執行一次以便註冊應用程式伺服器。接著在 Gordon 中執行用戶端應用程式，你可以在下圖中看到用戶端應用程式也啟動了 Gordon 之中的應用程式伺服器。然後我們使用用戶端應用程式異動一筆資料，並且按下 ApplyUpdates 按鈕藉由 Gordon 機器之中的應用程式伺服器異動回資料表之中。

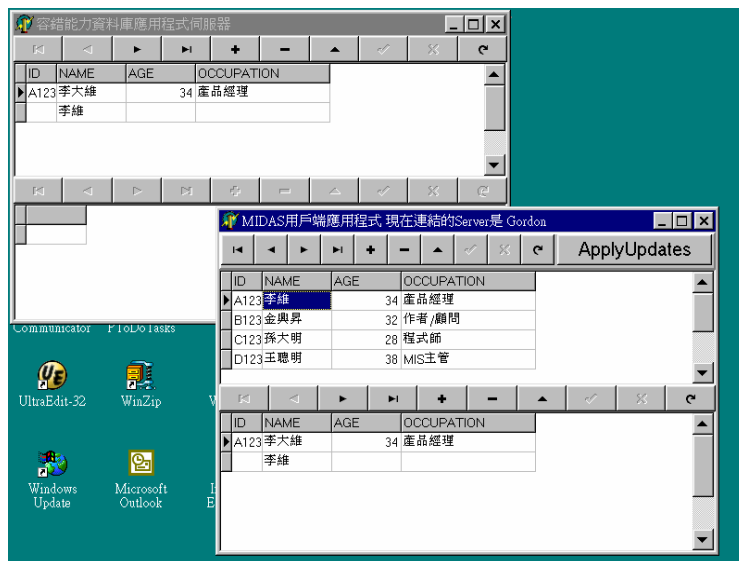


圖 5-39 範例程式執行的畫面。首先它藉由 Gordon 之中的應用程式伺服器異動資料

現在讓我們立刻如圖 5-40 所示強迫 Gordon 機器之中的應用程式伺服器結束執行。讓用戶端應用程式原先連結的應用程式伺服器無法繼續執行。

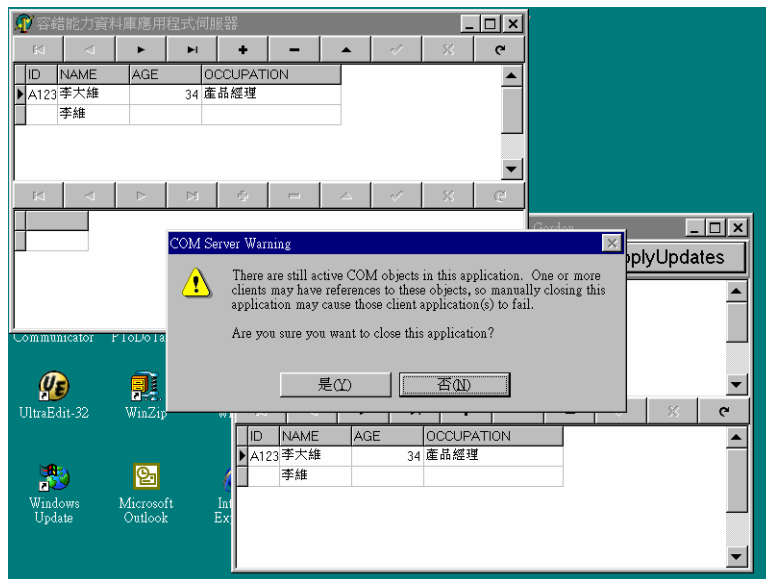


圖 5-40 現在我們強迫 Gordon 機器中的應用程式伺服器結束執行，讓用戶端應用程式產生例外

然後再讓我們在用戶端應用程式中異動另外一筆資料，並且按下表格中的 ApplyUpdates 按鈕試圖異動資料回資料表之中，如圖 5-41 所示：

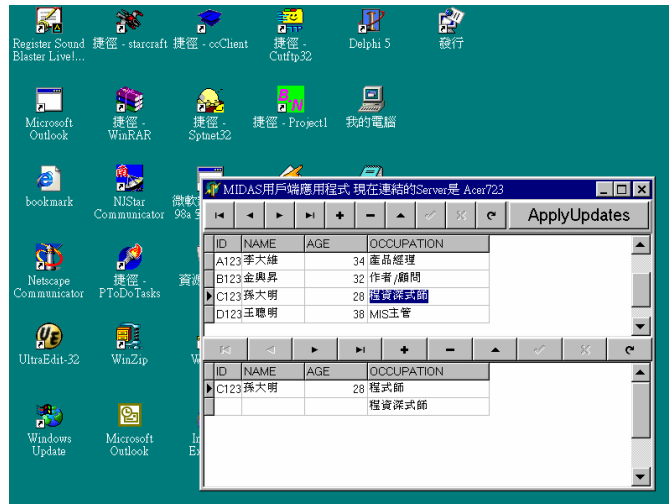


圖 5-41 用戶端應用程式再次異動一筆資料，但是重新連結到 Acer723 之中的應用程式伺服器異動資料，而不是原先的 Gordon 機器之中的應用程式伺服器

由於此時用戶端應用程式原先連結的應用程式伺服器已經故障了，所以它會試圖向 TSimpleObjectBroker 要求另外一台提供相同服務的應用程式伺服器，以便能夠順利的異動資料回資料表之中。由於此時還有 Acer723 機器之中的應用程式伺服器可以服務用戶端應用程式，所以 TSimpleObjectBroker 會自動啟動 Acer723 機器之中的應用程式伺服器，然後異動資料。圖 5-42 便是在 Acer723 機器之中的應用程式伺服器接受到用戶端應用程式異動的第三筆資料，並且異動回資料表的畫面。

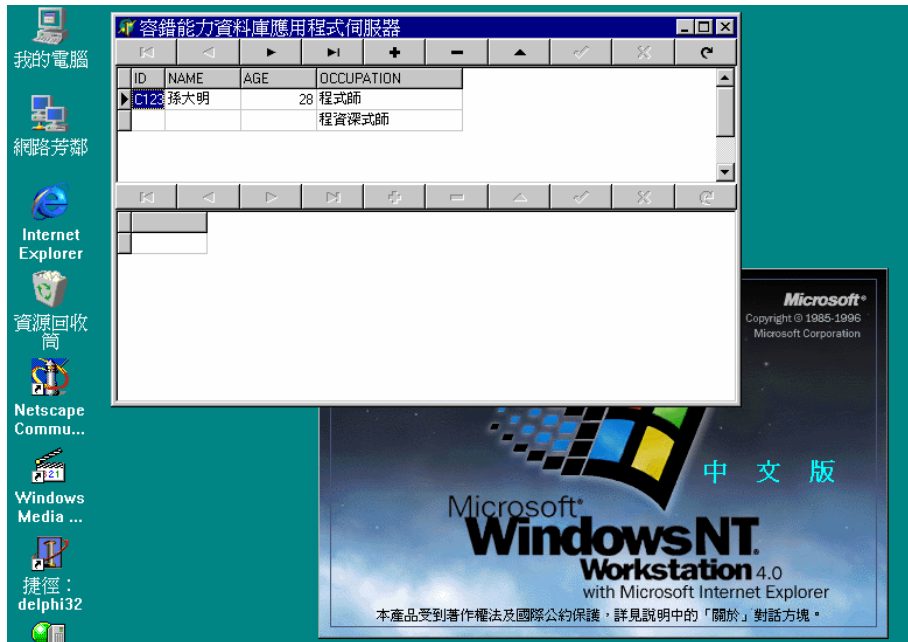


圖 5-42 Acer723 機器之中的應用程式伺服器果然顯示了用戶端應用程式傳遞過來的異動資料，並且更新異動資料回資料表之中

從上面的範例中可以證明我們的確開發出了一個非常安全強固的多層應用系統，這個應用系統即使是在原先連結的應用程式伺服器故障時仍然能夠自動找到另外一台應用程式伺服器把用戶端的異動資料更新回資料庫之中。

雖然如此，但是如果此時所有的應用程式伺服器都無法使用時，用戶端異動的資料該如何呢？在 5-5 小節中會繼續討論如何處理這種更為棘手的容錯問題。但是在討論之前，先讓我們再看看如何使用 CORBA 來開發安全強固的多層分散式應用系統。

5-6 更安全的容錯多層分散式應用系統

在前面的小節中使用了 COM/DCOM 技術實作安全強固的多層分散式應用系統。但是除了這些技術之外，是不是有更安全的應用系統呢？例如如果使用者在用戶端異動資料時，如果所有的應用程式伺服器都不幸故障了，那麼可

不可以讓用戶端的使用者能夠保存這些異動的資料，等到應用程式伺服器回復功能之後再把這些異動的資料更新回資料庫之中呢？

這個問題在前面的圖 5-27 中已經敘述了這些現象，現在的問題是程式師能不能夠實作出這種要求的多層分散式應用系統。事實上解決這個問題的答案幾乎已經存在了，那就是 TClientDataSet 的 SaveToFile 和 LoadFromFile 這兩個方法的觀念。如果在一個多層分散式應用系統中所有的應用程式伺服器都故障了，那麼只要用戶端的異動資料能夠暫時儲存異動的資料在硬式磁碟機之中，並且在應用程式伺服器回復功能時再從硬式磁碟機中讀取這些異動資料，再呼叫 ApplyUpdates 方法就可以把異動的資料更新回資料庫之中了。

但是為什麼 TClientDataSet 的 SaveToFile 和 LoadFromFile 方法能夠解決這個問題呢？我們可以使用一個簡單的範例就可以瞭解這兩個方法有什麼效果。首先使用 Delphi 建立一個簡單的應用程式伺服器並且連結的資料庫的 EMPLOYEE 資料表，然後建立如下的用戶端主表格。在這主表格中使用一個 TDCOMConnection 連結應用程式伺服器以及一個 TClientDataSet 顯示 EMPLOYEE 資料表。此外下方的 TDBGrid 和 TClientDataSet 是用來顯示修改 EMPLOYEE 資料表的 Delta 資料用的，這個技術在本書中已經使用了許多次所以就不再說明了。圖 5-43 是這個範例執行的畫面，此時由於沒有異動任何的資料所以表格下方的 TDBGrid 是空白的。接著我們修改圖 5-44 中前兩筆資料，此時圖 5-45 下方的 TDBGrid 就會顯示已經異動的資料。



圖 5-43 範例程式主表格

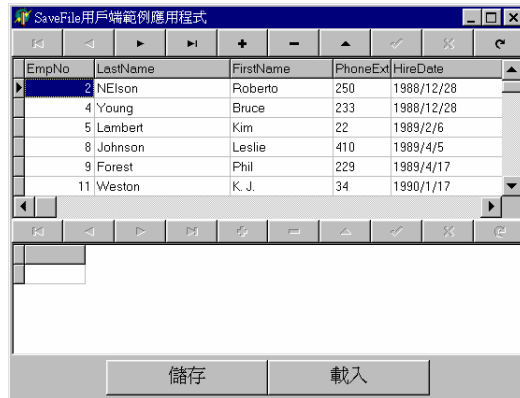


圖 5-44 範例程式執行時的畫面



圖 5-45 範例程式顯示 EMPLOYEE 資料表異動的 Delta 資料

現在讓我們點選表格中的『儲存』按鈕，然後你可以結束程式的執行，再重新執行這個範例程式，再點選『載入』按鈕，此時你可以看到圖 5-46 的畫面。請注意的是當程式碼呼叫 SaveToFile，再呼叫 LoadFromFile 方法時對於所有異動的過的資料 Delphi 都會新增在原資料集的最後面，此外 SaveToFile 方法會從應用程式伺服器取得資料表中所有的資料，然後才儲存在檔案之中，而且 SaveToFile 方法也會一併儲存所有異動的資料，也就是 TClientDataSet 的 Delta 資料。

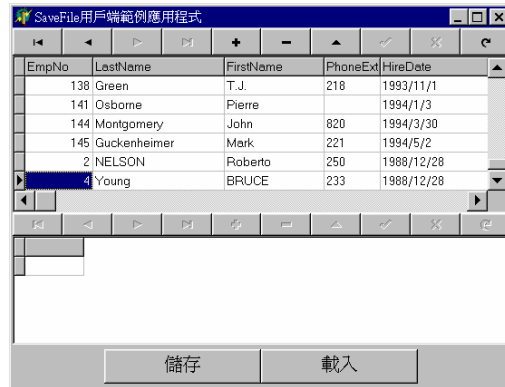


圖 5-46 按下『儲存』按鈕，然後再按下『載入』按鈕

請注意當我們呼叫 LoadFromFile 重新載入剛才異動的資料時，我們可以發現 SaveToFile 已經自動的合併了原始的資料，以及異的資料。下圖是 SaveToFile 以及 LoadFromFile 方法的執行流程圖：

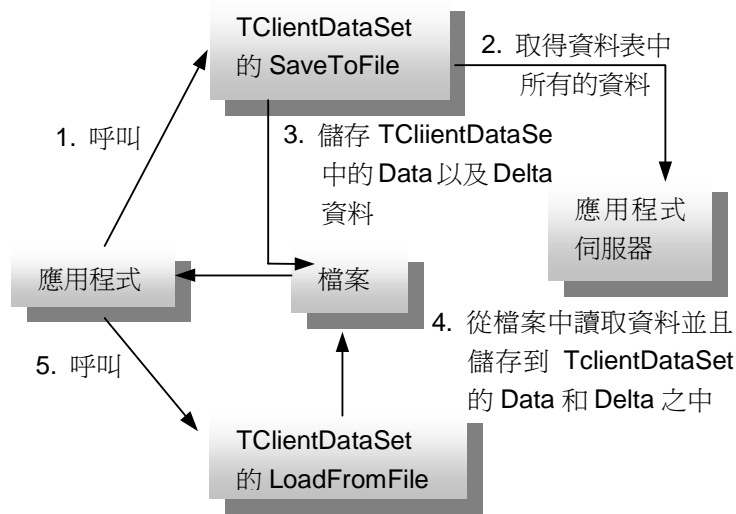


圖 5-47 SaveToFile 和 LoadFromFile 的執行流程

現在瞭解了 SaveToFile 和 LoadFromFile 這兩個方法的運作細節後，那麼要實作在所有應用程式伺服器都故障時用戶端應用程式仍然能夠運作的分散式應用程式就很簡單了。只要應用程式在應用程式伺服器故障時呼叫 SaveToFile 方法把資料表的資料以及在用戶端異動的資料儲存在檔案之中，然後在應用程式伺服器回復功能時再執行用戶端應用程式，呼叫 LoadFromFile 載入先前儲存的資料，最後再呼叫 ApplyUpdates 方法把用戶端的異動更新回資料表即

可。使用這種技術的分散式應用系統能夠在惡劣的情形下仍然能夠儲存異動的資料，等到執行環境回復正常之後再把資料更新回資料庫之中。

現在讓我們執行本章的一個範例程式看看這樣行不行的通。這個範例程式位於 Chap05\SaveFileDemoServer 目錄之下。當範例程式執行時我們先修改一筆資料如下圖所示：



圖 5-48 範例程式一執行的畫面，此時應用程式伺服器正常的執行，而且用戶端異動了 Roberto 這筆資料

然後先結束應用程式伺服器，讓用戶端應用程式無法更新資料回資料庫之中：

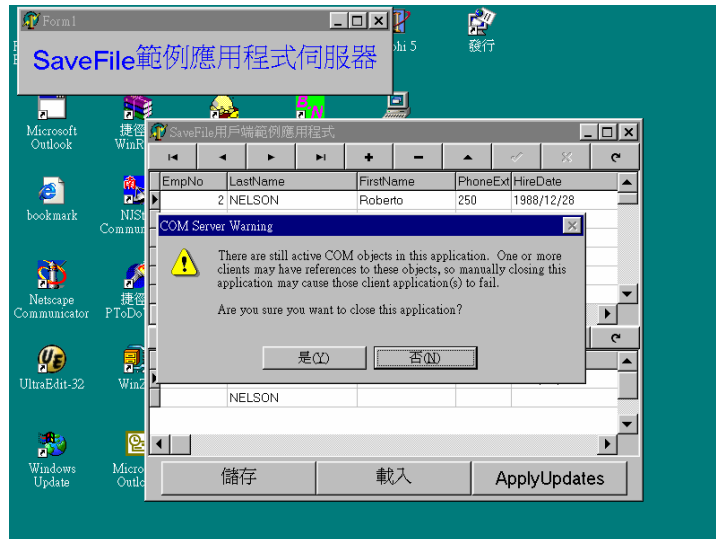


圖 5-49 結束執行應用程式伺服器

由於應用程式伺服器沒有執行，所以就點選表格中的儲存按鈕把用戶端的資料以及異動的 Roberto 這筆記錄儲存在檔案之中，然後再結束用戶端應用程式的執行。現在再度執行用戶端應用程式並且帶起應用程式伺服器，點選載入按鈕。從下圖中你可以看到原先的資料以及異動的 Roberto 記錄可以正確的載入到 TClientDataSet 之中。最後再點選表格中的 ApplyUpdates 按鈕把異動的 Roberto 記錄更新回資料庫之中。

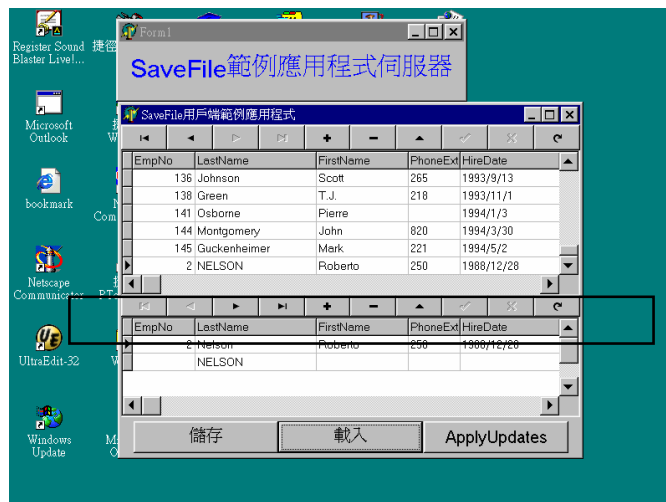


圖 5-50 再次執行範例程式並且點選表格中『載入』按鈕載入先前儲存的資料，

再點選 ApplyUpdates 按鈕更新資料回資料庫之中

最後再重新執行範例程式，從下圖中你可以看到異動的 Roberto 記錄果然正確的更新回資料表之中。

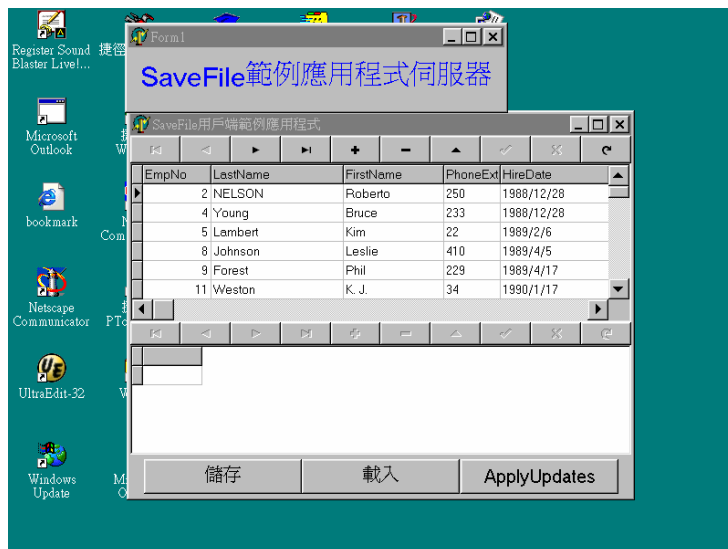


圖 5-51 重新執行範例程式，異動的資料果然更新回資料表之中了

從這個範例中你可以看到 Roberto 這筆異動的資料從線上異動，儲存到檔案之中，載入應用程式之中，最後再更新回資料表之中。在這個過程中應用程式伺服器從正常執行，到故障，再回復執行並且更新用戶端異動的資料回資料庫之中。這些所有的功能只使用了下列簡單的程式碼來完成的：

```

procedure TForm3.Button1Click(Sender: TObject);
begin
    ClientDataSet1.SaveToFile('Sample.DAT');
end;
procedure TForm3.Button2Click(Sender: TObject);
begin
    ClientDataSet1.LoadFromFile('Sample.Dat');
    ClientDataSet2.Data := ClientDataSet1.Delta;
end;
procedure TForm3.Button3Click(Sender: TObject);

```

```

begin
  ClientDataSet1.ApplyUpdates(0);
  try
    ClientDataSet1.Close;
    DCOMConnection1.Connected := False;
  finally
    DCOMConnection1.Connected := True;
    ClientDataSet1.Open;
  end;
end;

```

上面的程式碼使用了前面章節討論的觀念，TClientDataSet 的 Delta 特性中包含的異動資料能夠自由的更新回資料表之中，和它所連結的應用程式伺服器沒有什麼關係（除了 TClientDataSet 使用的資料 cursor 之外）。請注意，在上面的程式碼中當範例程式呼叫 ApplyUpdates 方法更新資料之後就再次的重新連結應用程式伺服器並且開啟 TClientDataSet 以便回復用戶端應用程式和應用程式伺服器之間正確的 cursor 狀態。

除了使用本節的觀念和技術之外，你可以結合 5-4 小節的技術讓這個範例也能夠隨意的連結區域網路或是廣域網路中提供相同服務的應用程式伺服器。

但是 Delphi 5 的 TClientDataSet 元件的 SaveToFile 方法有一個非常嚴重的缺點，那就是當程式碼呼叫 SaveToFile 時 SaveToFile 方法會向應用程式伺服器取得資料表中所有的資料，然後把這些資料表中所有的資料以及在用戶端應用程式中異動的資料儲存在檔案之中。這種行為對於分散式應用系統來說是非常不好的處理方式，因為如果此時資料表中有數 10 萬筆的資料，那麼 SaveToFile 會把這數 10 萬筆的資料從後端資料庫之中藉由應用程式伺服器取到用戶端儲存。這會大大的降低應用程式執行的效率，在廣域網路中這更是不能夠忍受的。

所以當你要使用本節的技術時就必須克服 SaveToFile 這個缺點。我們檢查 DBClient.Pas 程式單元就可以找到為什麼 TClientDataSet 的 SaveToFile 方法會取得並且儲存所有的資料。下面的程式碼顯示出了問題的所在：

```

procedure TClientDataSet.CheckProviderEOF;
begin
  if HasAppServer and not ProviderEOF and FFetchOnDemand and
    (FPacketRecords <> 0) then

```

```
FetchMoreData(True);
end;
procedure TClientDataSet.WriteDataPacket(Stream: TStream; WriteSize:
Boolean;
XMLFormat: Boolean = False);
var
Size: Integer;
DataPtr: Pointer;
begin
RCS;
if Active then CheckBrowseMode;
if IsCursorOpen then
begin
CheckProviderEOF;
SaveDataPacket(XMLFormat);
end;
if Assigned(FSavedPacket) then
begin
...
end;
end;

procedure TClientDataSet.SaveToStream(Stream: TStream; Format:
TDataPacketFormat = dfBinary);
begin
WriteDataPacket(Stream, False, (Format=dfXML));
end;

procedure TClientDataSet.SaveToFile(const FileName: string = '';
Format: TDataPacketFormat = dfBinary);
var
Stream: TStream;
begin
if FileName = '' then
Stream := TFileStream.Create(Self.FileName, fmCreate) else
Stream := TFileStream.Create(FileName, fmCreate);
try
if LowerCase(ExtractFileExt(FileName)) = '.xml' then
Format := dfXML;
SaveToStream(Stream, Format);
finally
```

向應用程式伺服器取得更多的資料

呼叫 CheckProviderEOF 向應用程式伺服器取得更多的資料

```

Stream.Free;
end;
end;

```

原來當程式碼呼叫了 SaveToFile 方法之後，SaveToFile 會呼叫 SaveToStream，SaveToStream 再呼叫 WriteDataPacket 方法。在 WriteDataPacket 中它會呼叫 CheckProviderEOF 方法，而 CheckProviderEOF 方法會呼叫 FetchMoreData。FetchMoreData 會不停的向應用程式伺服器取得資料，一直到資料表到達最後一筆記錄為止。

所以要克服 SaveToFile 的缺點，只要控制 TClientDataSet 呼叫 CheckProviderEOF 的時機就可以了。在 Chap05\SaveFileDemoServer 目錄之下有一個 FTClientDataSet 程式單元，FTClientDataSet 是包含了一個新的元件 TFTClientDataSet。這個元件定義了一個新的特性 SaveAllRecords。這個特性可以讓程式師控制 SaveToFile 是否要向應用程式伺服器取得所有的資料。SaveAllRecords 的內定值是 False，代表 TFTClientDataSet 元件在內定上只儲存已經存在的資料以及異動的資料，並不會向應用程式伺服器取得所有的資料。例如下圖便是在 Delphi 整合發展環境中使用 TFTClientDataSet 元件的情形。

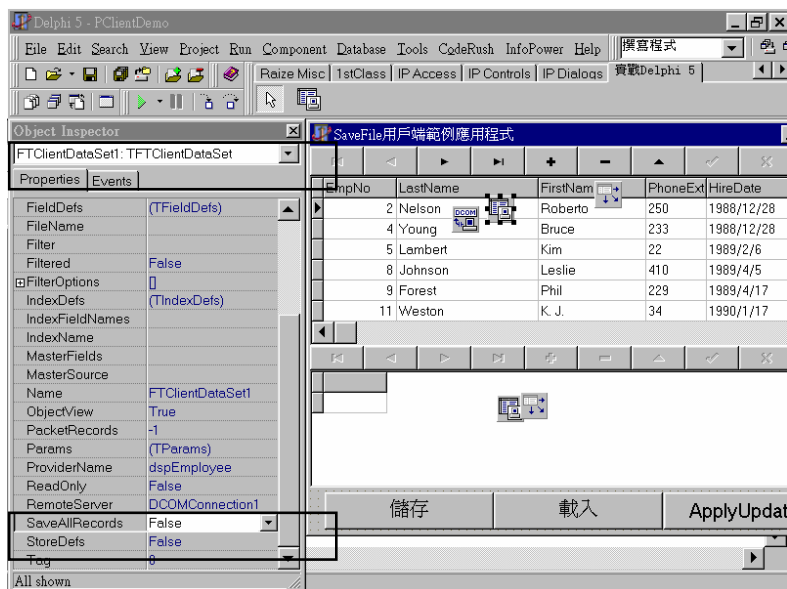


圖 5-52 TFTClientDataSet 元件有一個 SaveAllRecords 特性可以控制用戶端應用程式是否要取得並且儲存所有的記錄

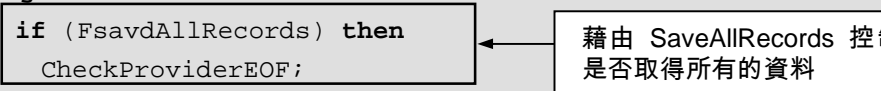
TFTClientDataSet 元件定義的行為不但比 TClientDataSet 元件更有效率，也比較合理。因為在直覺上當使用者儲存用戶端的資料時也只是希望儲存目前在用戶端的資料。

如果你有興趣看看 TFTClientDataSet 的實作程式碼，你可以在 Chap05\SaveClientDataSet 目錄下找到它。請注意 TFTClientDataSet 除了在 WriteDataPacket 方法中根據 SaveAllRecords 特性值控制是否取得所有的資料之外，TFTClientDataSet 也必須複載 (override) 許多 TClientDataSet 的方法，否則如果讓執行權進入 DBClient.PAS 的話執行的結果就不正確了。

```

procedure TFTClientDataSet.WriteDataPacket(Stream: TStream;
  WriteSize: Boolean);
var
  Size: Integer;
  DataPtr: Pointer;
begin
  RCS;
  if Active then CheckBrowseMode;
  if IsCursorOpen then
  begin
    if (FsavdAllRecords) then
      CheckProviderEOF;
    SaveDataPacket(XMLFormat);
  end;
  ...
end;

```



希望 TFTClientDataSet 能夠幫助你撰寫出更好的分散式應用系統，下面是 TFTClientDataSet 元件完整列表：

```

unit FTClientDataSet;

{$R-}

interface

uses Windows, SysUtils, ActiveX, Graphics, Classes, Controls,
  Forms, Db,
  BDE, DSIntf, DBCommon, StdVcl, DBClient;

```

```
type
  TFTPClientDataSet = class(TClientDataSet)
  private
    FSaveAllRecords : Boolean;
    FOpeningFile : Boolean;
  protected
    procedure WriteDataPacket(Stream: TStream; WriteSize: Boolean;
XMLFormat: Boolean = False);
    procedure CheckProviderEOF;
    procedure FetchMoreData(All: Boolean);
    procedure SaveDataPacket(XMLFormat: Boolean = False);
    procedure ClearSavedPacket;
    procedure SaveToStream(Stream: Tstream; Format:
TDataPacketFormat = dfBinary);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure SaveToFile(const FileName:string='TmpDataFile.dat';
Format: TDataPacketFormat = dfBinary);
    procedure LoadFromFile(const FileName:string='TmpDataFile.dat');
  published
    property SaveAllRecords : Boolean read FSaveAllRecords write
      SaveAllRecords default False;
  end;

procedure Register;

implementation

uses DBConsts, MidConst, ComObj, Provider, TypInfo;

procedure Register;
begin
  RegisterComponents('實戰 Delphi 5 ', [TFTPClientDataSet]);
end;

constructor TFTPClientDataSet.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
```

```
    FSaveAllRecords := False;
    FopeningFile := False;
end;

destructor TFTClientDataSet.Destroy;
begin
    inherited Destroy;
end;

procedure TFTClientDataSet.FetchMoreData(All: Boolean);
var
    Count: Integer;
    RecsOut: Integer;
begin
    if All then Count := AllRecords else Count := PacketRecords;
    if Count = 0 then Exit;
    AddDataPacket(DoGetRecords(Count, RecsOut, 0, '', Unassigned),
RecsOut <> Count);
    ProviderEOF := RecsOut <> Count;
end;

procedure TFTClientDataSet.CheckProviderEOF;
begin
    if HasAppServer and not ProviderEOF and FetchOnDemand and
(PacketRecords <> 0) then
        FetchMoreData(True);
end;

procedure TFTClientDataSet.SaveDataPacket;
const
    StreamMode: array[Boolean] of DWord = (xmlOFF, xmlON);
var
    DataPacket: TDataPacket;
begin
    DataPacket := VarToDataPacket(Data);
    if Assigned(DSBase) and (DataSetField = nil) then
        begin
            DSBase.SetProp(dspropXML_STREAMMODE, StreamMode[XMLFormat]);
            ClearSavedPacket;
            Check(DSBase.StreamDS(DataPacket));
        end;
end;
```

```
end;

procedure TFTClientDataSet.ClearSavedPacket;
var
    DataPacket: TDataPacket;
begin
    DataPacket := VarToDataPacket(Delta);
    FreeDataPacket(DataPacket);
end;

procedure TFTClientDataSet.WriteDataPacket(Stream: TStream;
    WriteSize: Boolean; XMLFormat: Boolean = False);
var
    Size: Integer;
    DataPtr: Pointer;
begin
    if Active then
        begin
            CheckBrowseMode;
            if (FSaveAllRecords) then
                CheckProviderEOF;
                SaveDataPacket(XMLFormat);
            end;
            if Assigned(VarToDataPacket(Data)) then
                begin
                    Size := DataPacketSize(VarToDataPacket(Data));
                    SafeArrayAccessData(VarToDataPacket(Data), DataPtr);
                    try
                        if WriteSize then
                            Stream.Write(Size, SizeOf(Size));
                            Stream.Write(DataPtr^, Size);
                        finally
                            SafeArrayUnAccessData(VarToDataPacket(Data));
                        end;
                        if Active then ClearSavedPacket;
                    end;
                end;
            end;
        end;

procedure TFTClientDataSet.SaveToStream(Stream: Tstream; Format:
TDataPacketFormat = dfBinary);
begin
    WriteDataPacket(Stream, False, (Format=dfXML));
```

```
end;  
  
procedure TFTClientDataSet.SaveToFile(const FileName: string; Format:  
TDataPacketFormat = dfBinary);  
var  
    Stream: TStream;  
begin  
    if FileName = '' then  
        Stream := TFileStream.Create(Self.FileName, fmCreate) else  
        Stream := TFileStream.Create(FileName, fmCreate);  
    try  
        if LowerCase(ExtractFileExt(FileName)) = '.xml' then  
            Format := dfXML;  
        SaveToStream(Stream, Format);  
    finally  
        Stream.Free;  
    end;  
end;  
  
procedure TFTClientDataSet.LoadFromFile(const FileName: string);  
var  
    Stream: TStream;  
begin  
    Close;  
    if FileName = '' then  
        Stream := TFileStream.Create(Self.FileName, fmOpenRead) else  
        Stream := TFileStream.Create(FileName, fmOpenRead);  
    try  
        FOpeningFile := True;  
        try  
            LoadFromStream(Stream);  
        finally  
            FOpeningFile := False;  
        end;  
    finally  
        Stream.Free;  
    end;  
end;  
end.
```

5-7 結論

本章延續了第 4 章節對於 MIDAS 以及 Delphi 5 多層應用系統功能的介紹。在本章中除了討論一些 MIDAS 更進階的觀念之外，也使用對於 MIDAS 的深入瞭解來開發具備強勁功能的容錯多層分散式應用系統。

無狀態物件是 MIDAS 3.0 強調的新功能，也是 MTS 等中介軟體要求程式師使用的物件種類。但是在許多的應用程式中我們都需要維護狀態資訊，那麼程式師如何在無狀態物件中為應用程式維護狀態資訊便成為非常重要的功能，本章說明了如何在用戶端應用程式中為中介物件維護資訊，而讓中介的物件能夠成為無狀態物件，進而解決了許多應用程式的需求。在『實戰 Delphi 5.x-分散式 Web 應用系統篇』一書中將會繼續的討論其他為無狀態物件維護狀態資訊的技術。

在程式師開發多層分散式應用系統時，可以使用 Delphi 5 的 TSimpleObjectBroker 元件註冊所有能夠執行應用程式伺服器的機器，然後讓用戶端應用程式執行時動態的從其中選擇一個特定機器的應用程式伺服器來連結並且使用。而且當用戶端應用程式在執行的中途原先連結的應用程式伺服器故障後，能夠和 TSimpleObjectBroker 互動，進而取得另外一台能夠提供相同服務的應用程式伺服器再繼續執行。

Delphi 5 除了允許程式師配合 COM/DCOM 技術開發具備容錯能力的多層分散式應用系統之外，也允許程式師使用 CORBA 的技術開發出能夠整合異質平台的多層分散式應用系統，在本書的第 11，12 章將會對 CORBA 做詳細的說明，並且展示如何使用 CORBA 開發分散式應用系統。

除了一般可能發生的問題之外，當分散式應用系統執行時可能會發生所有的應用程式伺服器都因為某些原因而無法提供用戶端應用程式服務。在這種情形之下，程式師應該開發出允許使用者暫時儲存已經在用戶端異動的資料，並且在有任何的應用程式伺服器回復功能時能夠再向 TSimpleObjectBroker 元件要求回傳一個可以執行的應用程式伺服器，然後藉由這個回傳的應用程式伺服器異動資料。要做到這種能力的容錯能力應用系統，程式師必須繼承用戶端使用的 TClientDataSet 元件，並且定義新的方法讓使用者能夠同時儲存用戶端的

原始資料以及異動的資料到硬式磁碟機之中，等到有應用程式伺服器回復功能時再載入異動的資料，並且藉由這台應用程式伺服器把資料異動回資料庫之中。在 5-5 小節中開發了一個類似 TClientDataSet 元件的後代類別，這個新的類別能夠同時儲存原始資料以及異動的資料，並且具備再載入這些儲存資料的能力。程式師可以使用這個新的元件開發出更為安全，可靠以及具備強勁容錯能力的分散式應用系統。

希望本章討論的內容以及技術能夠讓你瞭解處理容錯能力的觀念和技巧，更希望你能夠藉由本章討論的技術開發出更為先進，完善的容錯多層分散式應用系統。