소켓 프로그래밍 기법의 활용

(Using Socket Programming Techniques)

윈도우에서의 프로세스간 통신 기법으로는 명명된 파이프, DCOM, DDE, 클립 보드와 각종 네트워크 프로그래밍 기법 등을 이용할 수 있다. 이 중에서도 윈도우 95 와 윈도우 NT 3.5 버전부터는 내부적인 통신 프로토콜로 기존의 NetBIEU 와 함께 TCP/IP 를 사실상의 표준으로 인정하고 이를 지원하고 있다. 또한, DCOM 과 윈도우 소켓을 프로세스간 통신의 표준으로 삼고 있으며, 윈도우 NT 4.0 부터는 윈도우 소켓의 2.0 버전(WinSock 2.0)을 사용하여 보다 강화된 소켓 프로그래밍을 지원하게 되었다.
이러한 소켓 프로그래밍을 위해서는 Win32 에서 지원하는 API 를 직접 이용하여 프로그래밍을 할 수도 있겠으나, 델파이에서 지원하는 소켓 컴포넌트를 이용하면 쉽게 소켓을 지원하는 어플리케이션을 지원할 수 있다.
이번 장에서는 소켓 컴포넌트를 이용하는 방법과 소켓 프로그래밍 기법을 익혀보도록 한다.

소켓 프로그래밍의 기초

소켓이란 네트워크 프로토콜을 구현할 때 여기에 저장된 데이터를 포함한 커다란 집합에 대한 핸들이다. 쉽게 말하면 네트워크에 대한 파일 핸들이라고 생각하면 된다. 네트워크 프로그래밍의 기본은 TCP/IP 연결을 하고, 소켓을 생성해서 이를 전송하거나 받는 것이다.
소켓에는 원래 서버용, 클라이언트용 소켓이 분리되어 있는 것은 아니다. 그렇지만 델파이에서는 TServerSocket, TClientSocket 으로 분리된 소켓을 제공하고 있다. 이들은 모두 TCustomSocket 클래스에서 상속받은 것으로 내부적으로는 동일한 소켓을 사용하고, 사용방법도 거의 비슷하다.
서버가 되는 소켓은 클라이언트측 소켓과는 달리 클라이언트의 연결 요구를 기다리는 listen 이라는 작업을 해야하고, 클라이언트는 서버에 연결(connect)해야 한다. 여기서 양측 소켓의 Active 프로퍼티를 True 로 설정하면 서버와 클라이언트가 접속된다. 또한, TServerSocket 클래스에는 여러 개의 클라이언트가 접속되었을 때 이를 관리할 수 있는 기능을 추가로 가지고 있다.

IP 주소와 포트

델파이의 소켓 컴포넌트에는 Address 와 Port 라는 프로퍼티가 있다. Address 프로퍼티는 IP 주소를 나타내며, Port 프로퍼티는 서버로 들어오는 메시지를 통과시키는 번호이다. 포트가 없으면 하나의 서버 컴퓨터에는 하나의 서버 프로그램만 설치할 수 있다. 포트 번호

를 이용해서 동일한 서버 컴퓨터에 여러 개의 서버 프로그램을 사용할 수 있다. 예를 들어 메일 서버, 뉴스그룹 서버, 웹서버, FTP 서버, 텔넷 서버 등의 프로그램들은 모두 다른 포트를 사용한다. 이런 포트 번호 중 일반적으로 사용되는 서비스에 관한 것들이 있는데 SMTP 는 25, NNTP 는 119, Telnet 은 23, FTP 는 21 을 보통 사용한다.

소켓과 소켓 연결 (Socket Connection)

소켓은 네트워크 어플리케이션이 네트워크 상의 다른 시스템 사이를 통신할 수 있도록 도와주는 도구가 된다. 각각의 소켓은 하나의 네트워크 연결로 생각할 수 있는데, 여기에는 어플리케이션이 실행되는 시스템, 인터페이스 종류, 연결에 사용된 포트에 대한 주소가 있어야 한다. 그러므로, 소켓 연결에 대해서 충분히 알기 위해서는 각 연결 부분에 대한 소켓의 주소를 반드시 알아야 한다.
이렇게 소켓 연결을 하기 전에 연결 부분을 담당하게 되는 소켓에 대한 정보를 제공해야 하는데, 일부의 정보는 어플리케이션이 실행되고 있는 시스템에서 알아낼 수 있다. 예를 들어, 각 클라이언트 소켓의 로컬 IP 주소에 대한 정보는 운영체제에서 알아낼 수 있으므로 따로 제공할 필요가 없다.
따로 제공해야 하는 정보는 현재 작업하고 있는 소켓의 종류에 따라 달라지는데, 클라이언트 소켓은 연결하고자 하는 서버에 대한 정보를 제공해야 하며 서버 소켓은 제공하는 서비스를 제공하는 포트에 대한 정보를 제공해야 한다. 이러한 소켓 연결에 대한 정보에는 IP 주소와 포트 번호가 모두 포함된다.

호스트(Host)란 ?

호스트는 소켓을 포함한 어플리케이션이 동작하는 시스템을 말한다. 이렇게 소켓에 대한 호스트를 지정할 때에는 다음과 같이 표준 인터넷 주소로 사용되는 IP 주소 표기 방식을 많이 사용한다.

123.123.1.2

하나의 시스템은 하나 이상의 IP 주소를 지원하게 된다. IP 주소는 기억하기도 어렵거니와 알아보기도 쉽지 않기 때문에, 호스트의 이름을 지정하는 방식을 같이 사용하여 이러한 단점을 극복한다. 이렇게 이름으로 된 방식의 IP 주소에 대한 앨리어스를 URLs(Uniform Resource Locators) 라고 하며, 다음과 같은 도메인 이름과 서비스를 포함한 형태가 된다.

http://www.ExamSite.Com

서버 소켓은 시스템에서 로컬 IP 주소를 알아낼 수 있기 때문에 호스트를 지정할 필요가 없다. 만약 로컬 시스템이 하나 이상의 IP 주소를 가지고 있을 경우에는 서버 소켓은 모든 IP 주소에 대한 클라이언트의 요구를 이용한다. 서버 소켓이 연결되면 클라이언트 소켓은 리모트 IP 주소를 제공하게 된다. 클라이언트 소켓은 반드시 호스트 이름이나 IP 주소를 입력해서 리모트 호스트를 지정해 주어야 한다.

연결의 종류

소켓 연결에는 연결의 초기화와 어떤 로컬 소켓이 연결되는지에 따라 기본적으로 다음과 같은 세가지로 나누어 볼 수 있다.

1. 클라이언트 연결 (Client connections)

클라이언트 연결은 로컬 시스템의 클라이언트 소켓을 리모트 시스템의 서버 소켓에 연결하는 것을 말한다. 클라이언트 연결은 클라이언트 소켓에 의해 개시되고 초기화된다. 먼저 클라이언트 소켓이 연결하고자 하는 서버 소켓에 대한 정보를 제공하면, 클라이언트 소켓이 서버 소켓을 찾게 되고, 서버의 위치를 파악하게 되면 연결을 요구한다. 서버 소켓은 클라이언트 요구에 대한 큐(queue)를 가지고 있어서 시간이 될 때마다 연결을 시도한다. 일단 서버 소켓이 클라이언트 연결을 받아들이면 클라이언트 소켓에 연결된 서버 소켓에 대한 모든 정보를 전송하게 되며, 클라이언트에 의해 연결이 완료된다

2. 리스닝 연결 (Listening connections)

서버 소켓이 활동적으로 클라이언트를 찾아서 연결을 시도하지 않고, 수동적으로 클라이언트의 요구를 기다리는 하프 연결 (half connection) 상태를 유지하는 형태의 연결이다. 서버 소켓은 큐를 리스닝 연결과 연관지어서 관리하며, 큐에는 클라이언트의 연결 요구가 계속 기록된다. 일단 서버 소켓이 클라이언트의 연결 요구를 받아들이면 클라이언트와 연결하기 위한 새로운 소켓을 생성하게 된다.
이렇게 하면 리스닝 연결 자체는 다른 클라이언트 요구를 받아들일 수 있도록 계속 열려있게 할 수 있다.

3. 서버 연결 (Server connections)

서버 연결은 서버 소켓에 의해서 이루어지는 것으로, 리스닝 소켓이 클라이언트 요구를 받아들이면 생성된다. 일단 서버가 연결을 받아들이면 서버 소켓의 정보가 클라이언트에게 전송되어 클라이언트 소켓이 이 정보를 받으면 연결이 완료되는 형태이다.

일단 연결이 되면 서버 연결과 클라이언트 연결은 차이가 없는 연결 방식이다. 기본적으로 클라이언트 연결과 서버 연결은 두개의 종료점(endpoint)를 가지며 같은 능력과 같은 종류의 이벤트를 사용한다. 그에 비해 리스닝 연결은 단지 하나의 종료점 만을 가지고 있는 본질적으로 다른 연결방식이다.

서비스 프로토콜

네트워크 서버와 클라이언트를 개발하기에 앞서, 먼저 어플리케이션이 제공하거나 사용하게 될 서비스에 대한 이해가 반드시 선행되어야 한다. 많은 서비스 들은 네트워크 어플리케이션이 반드시 지원해야 하는 표준 프로토콜을 가지고 있다. 만약 HTTP, FTP 등의 표준 서비스를 지원하는 네트워크 어플리케이션을 제작한다면, 다른 시스템과 통신하게 되는 프로토콜에 대한 이해가 필요하다.
만약 다른 시스템과 통신하는데 있어서 새로운 형태의 서비스를 제공한다면, 제일 먼저 서비스를 사용하게 될 서버와 클라이언트 사이의 통신 프로토콜을 디자인해야 한다. 이때에는 어떤 메시지가 전달될 것이며, 이런 메시지 들이 어떻게 조화를 이루어야 하며, 정보의 암호화는 어떤 형식으로 할 것인지 등을 결정해야 한다.
가끔 네트워크 서버와 클라이언트 어플리케이션에서 네트워킹 소프트웨어와 서비스를 사용하는 어플리케이션 사이에 레이어(layer)를 제공하는 경우가 있다. 예를 들어, HTTP 서버는 인터넷과 웹서버 어플리케이션 사이에 위치하여 컨텐트를 제공하고, HTTP 리퀘스트 메시지에 반응하게 된다.
소켓은 네트워크 서버와 클라이언트 어플리케이션, 네트워킹 소프트웨어 사이에 인터페이스를 제공한다. 이때 ISAPI 등의 흔히 사용되는 표준 서버에 대한 API 를 복사해서 사용할 수도 있고, 자신만의 API 를 디자인해서 사용할 수도 있다.

서비스와 포트

대부분의 서비스는 특정 포트 번호와 연관되어 있다. 이럴 때에는 포트 번호를 서비스에 대한 번호 코드(numeric code)로 생각할 수 있다. 만약 표준 서비스를 구현한다면 델파이의 윈도우 소켓 객체의 메소드를 이용하면 그 서비스에 대한 포트 번호를 알아낼 수 있다. 그에 비해 새로운 서비스를 제공하는 경우라면 윈도우 95 나 NT 의 Services 파일에 포트 번호를 연관시켜 지정할 수 있다.

소켓 연결과 데이터 송수신

다른 시스템과 소켓 연결을 하는 이유는 연결을 통해서 정보를 송수신할 수 있기 때문이다. 이때 어떤 정보를 주고 받을 지와 언제 어떤 방식을 사용할 지 등은 소켓 연결에 사용된 서

비스에 좌우된다.
이렇게 데이터를 읽고 쓰는 데에는 두가지 방식이 있다. 소켓에 데이터를 읽고 쓸 때 비동기적인 방식을 사용해서 네트워크 어플리케이션에서 다른 코드의 실행을 방해하지 않는 것을 논-블로킹 연결(Non-Blocking connections)이라고 하며, 데이터를 읽고 쓰는 작업을 쓰레드를 이용하여 독립적으로 실행하는 형태의 연결을 블로킹 연결(Blocking connections)이라고 한다.

블로킹 연결 (Blocking connections)

클라이언트 소켓에서는 ClientType 프로퍼티를 ctBlocking 으로 설정하면 블로킹 연결이 생성된다. 클라이언트 어플리케이션에 따라서는 읽고 쓰는 데에 새로운 쓰레드를 생성하기를 원할 수도 있는데, 이렇게 하면 어플리케이션은 연결이 완료되어 데이터를 읽고, 쓸 때까지 다른 쓰레드를 실행할 수 있다.
서버 소켓에서는 ServerType 프로퍼티를 stThreadBlocking 으로 설정하면 블로킹 연결이 생성된다. 블로킹 연결은 연결에 의한 데이터 교환이 될 때까지 실행이 되지 않으므로, 다른 클라이언트 연결에 대해서 항상 새로운 쓰레드가 생성된다.

- 블로킹 연결과 쓰레드의 이용

클라이언트 소켓은 블로킹 연결이 사용될 때 새로운 쓰레드가 자동으로 생성되지 않는다. 만약, 클라이언트 어플리케이션이 데이터를 읽고, 쓰는 것 이외에 다른 작업이 없다면 이러한 방식도 큰 상관이 없겠지만, 연결이 이루어지지 않은 경우에도 사용자 인터페이스에서 다른 작업을 할 수 있게 하려면 새로운 쓰레드를 생성해야 한다.
그에 비해 서버 소켓은 블로킹 연결이 생성될 때마다 각 클라이언트 연결에 대해 새로운 쓰레드가 생성된다. 그렇게 때문에 연결을 통해 클라이언트가 데이터를 읽고 쓰는 작업을 할 때 다른 클라이언트가 이를 기다리지 않아도 된다. 서버 소켓은 TServerClientThread 객체를 사용해서 각 연결에 대한 쓰레드를 구현한다. 일단 서버 클라이언트 쓰레드가 실행되면, 연결되어 있는 클라이언트 측에서 연결을 통해 데이터를 쓰고 있는지 검사하여 그렇다면 OnClientRead 이벤트를 발생시키며, 그렇지 않으면 OnClientWrite 이벤트를 발생시키고 서버가 데이터를 쓰기 시작한다.

- TWinSocketStream 클래스의 활용

논-블로킹 연결과 블로킹 서버 연결 모두 연결에 의해 데이터를 읽고, 쓸 수 있게 되면 특정 이벤트를 받게 된다. 이때 이벤트 핸들러에서는 실제로 데이터를 읽고, 쓰는 작업을 윈도우 소켓 객체의 메소드를 이용해서 실행한다.

블로킹 클라이언트 연결에서는 반드시 자기 자신이 반대편의 연결이 데이터를 읽고, 쓸 준비가 되었는지를 파악해야 한다. 이때 TWinSocketStream 객체를 이용해서 연결을 통한 데이터 읽기, 쓰기를 실행하면, 적절한 시간 간격 등을 조율할 수 있는 메소드를 제공받을 수 있다. 예를 들어, WaitForData 메소드를 호출하면 서버 소켓이 준비될 때까지 기다리게 할 수 있다.

- 클라이언트 쓰레드의 작성

클라이언트 접속을 위한 쓰레드를 작성할 때에는 New Thread Object 대화 상자를 이용하여 새로운 쓰레드 객체를 정의할 수 있다. 새로운 쓰레드 객체의 Execute 메소드는 실제 쓰레드 연결을 통한 데이터 읽기와 쓰레드를 관리한다. 다음의 코드가 전형적인 클라이언트 쓰레드의 예이다.

```
procedure TMyClientThread.Execute;
var
  TheStream: TWinSocketStream;
  buffer: string;
begin
  TheStream := TWinSocketStream.Create(ClientSocket1.Socket, 60000);
          //TWinSocketStream 을 생성한다.
  try
    while (not Terminated) and (ClientSocket1.Active) do
    begin
      try
        GetNextRequest(buffer);
        TheStream.Write(buffer, Length(buffer) + 1);     //버퍼의 내용을 서버에 기록한다.
        ...
      except
        if not(ExceptObject is EAbort) then
          Synchronize(HandleThreadException);
      end;
    end;
  finally
    TheStream.Free;
  end;
end;
```

쓰레드를 이용하려면 OnConnect 이벤트 핸들러를 작성하면 된다.

- 서버 쓰레드의 작성

서버 접속에 대한 쓰레드는 TServerClientThread 에서 상속받는다.
서버 쓰레드를 구현하기 위해서는 Execute 메소드가 아닌, ClientExecute 메소드를 오버라이드해야 한다. 클라이언트 쓰레드와 비슷하게 구현하지만, 클라이언트 소켓 컴포넌트 대신에 TServerClientWinSocket 객체를 사용한다는 점이 가장 큰 차이점이다. 이 객체는 ClientSocket 프로퍼티를 통해 접근이 가능하다. 그리고, 예외 처리는 HandleException 메소드를 호출하면 된다.

```
procedure TMyServerThread.ClientExecute;
var
  Stream: TWinSocketStream;
  Buffer: array[0 .. 9] of Char;
begin
  while (not Terminated) and ClientSocket.Connected do
  begin
    try
      Stream := TWinSocketStream.Create(ClientSocket, 60000);
      try
        FillChar(Buffer, 10, 0);                    //버퍼를 초기화 한다.
        if Stream.WaitForData(60000) then
        begin
          if Stream.Read(Buffer, 10) = 0 then ClientSocket.Close;
                                                    //1 분까지 기다린다.
          ...
        end
        else
          ClientSocket.Close;
      finally
        Stream.Free;
      end;
    except
      HandleException;
```

```
    end;
  end;
end;
```

이 쓰레드를 사용하려면 OnGetThread 이벤트 핸들러에서 쓰레드를 생성하면 된다.

논-블로킹 연결 (Non-Blocking connections)

클라이언트 소켓에서 ClientType 프로퍼티를 ctNonBlocking 으로 설정하면 논-블로킹 연결이 이루어 진다. 이 경우 반대 편의 서버가 데이터를 읽거나 쓰게 되면 클라이언트 소켓이 이를 알 수 있으며, 이때 OnRead 또는 OnWrite 이벤트 핸들러에 의해서 반응하게 된다.
서버 소켓에서는 ServerType 프로퍼티를 stNonBlocking 으로 설정하면 논-블로킹 연결이 생성되는데, 논-블로킹 클라이언트 연결처럼 클라이언트에서 데이터를 읽고 쓰게 되면 OnClientRead 또는 OnClientWrite 이벤트 핸들러에 의해서 반응하게 된다.
이러한 이벤트에서 소켓 연결과 관련한 윈도우 소켓 객체가 파라미터로 전달되며, 이들 객체를 이용하면 여러가지 메소드를 활용할 수 있다. 소켓 연결에서 데이터를 읽을 때에는 ReceiveBuf, ReceiveText 메소드를 사용할 수 있다. ReceiveBuf 메소드는 사용하기 전에 ReceiveLength 메소드를 사용해서 반대 편 연결에서 전송하려는 데이터의 바이트 수를 결정한 후에 사용한다.
SendBuf, SendStream, SendText 메소드 등을 이용하면 데이터를 쓸 수 있다. 또한, 데이터를 쓰고 나서 더 이상 소켓 연결을 유지할 필요가 없을 때에는 SendStreamThenDrop 메소드를 사용해서 스트림에서 읽은 데이터를 모두 전송하고 연결을 닫도록 할 수 있다.
참고로 SendStream, SendStreamThenDrop 메소드를 사용하면 소켓이 연결이 종료된 후 스트림을 자동으로 메모리에서 해제하므로, 스트림 객체를 직접 해제할 필요가 없다.

클라이언트 소켓의 이용

어플리케이션을 TCP/IP 클라이언트로 사용할 때에는 클라이언트 소켓 컴포넌트(TClientSocket)를 폼이나 데이터 모듈에 추가한다. 클라이언트 소켓에는 연결하고자 하는 서버 소켓과 서버에서 제공받을 서비스를 지정하게 된다.
각각의 클라이언트 소켓 컴포넌트는 연결에서의 클라이언트측 종료점을 나타내게 되는 클라이언트 윈도우 소켓 객체(TClientWinSocket)을 사용한다.

- 서버의 지정

클라이언트 소켓 컴포넌트는 서버 시스템과 연결하고자 하는 포트를 지정할 때 사용할 수

있는 프로퍼티가 있다. 서버 시스템은 Host 프로퍼티에서 호스트의 이름을 이용해서 지정하거나, Address 프로퍼티에 직접 IP 주소를 적어넣을 수 있다. 만약 둘 다 지정한 경우에는 호스트 이름을 사용한다.
포트 역시 Port 와 Service 프로퍼티를 이용해서 지정할 수 있는데, Port 프로퍼티에는 포트의 번호를 직접 지정하는 것이고 Service 프로퍼티에는 포트 번호와 연관된 표준 서비스의 이름을 지정할 경우 간접적으로 포트가 지정되는 것이다. 둘 다 지정된 경우에는 서비스 이름을 사용한다.

- 연결의 생성

연결하고자 하는 서버에 대한 정보를 클라이언트 소켓 컴포넌트에 설정하고 나면, 런타임에서 Open 메소드를 호출하여 연결을 시도하게 된다. 디자인 시에 Active 프로퍼티를 True로 설정하면 어플리케이션이 시작할 때 연결을 시도한다.

- 연결에 대한 정보 얻기

서버 소켓과의 연결이 완료되면 클라이언트 윈도우 소켓 객체를 사용해서 연결에 대한 여러 가지 정보를 얻어올 수 있게 된다. 이때 이 객체를 얻어오기 위해 Socket 프로퍼티를 사용한다. 윈도우 소켓 객체에는 클라이언트와 서버 소켓이 사용하는 포트 번호와 주소를 결정할 때 사용하는 프로퍼티가 있으며, SocketHandle 프로퍼티를 이용해서 윈도우 소켓 API를 호출할 때 사용할 소켓 연결에 대한 핸들을 얻을 수도 있다. 또한, Handle 프로퍼티를 이용해서 소켓 연결에서의 메시지를 받는 윈도우에 접근할 수 있으며 ASyncStyles 프로퍼티를 이용해서 윈도우 핸들이 받게 되는 메시지의 종류를 결정할 수도 있다.

- 연결 종료

서버 어플리케이션과의 소켓 연결을 통한 통신이 끝나면, Close 메소드를 이용해서 연결을 종료할 수 있다. 이때 서버 측에서 연결을 종료하면 OnDisconnect 이벤트가 발생하므로, 적절한 처리를 해줄 수 있다.

서버 소켓의 이용

어플리케이션을 TCP/IP 서버로 둔갑시키려면 먼저 서버 소켓 컴포넌트인 TServerSocket을 폼이나 데이터 모듈에 올려 놓는다. 서버 소켓에서 제공하려는 서비스나 클라이언트의 요구를 기다릴 때 사용할 포트를 지정할 수 있다. 각 서버 소켓 컴포넌트는 서버 윈도우 소켓 객체(TServerWinSocket)를 사용하여 리스닝 연결에서의 서버측 종료점을 이루게 한

다. 또한, 서버가 받아들인 클라이언트 소켓과의 연결에서의 서버 종료점에 대한 클라이언트 윈도우 소켓 객체(TServerClientWinSocket)도 활용한다.

- 포트의 지정

서버 소켓이 클라이언트의 요구를 기다리기 전에 (이런 기다림을 'listening' 이라고 한다.) 서버가 사용할 포트를 지정해 주어야 한다. 이때 Port 프로퍼티를 사용해서 포트를 지정할 수 있다. 서버 어플리케이션이 특정 포트 번호와 연관된 표준 서비스를 제공하는 경우라면 Service 프로퍼티를 지정함으로써 간접적으로 포트를 지정할 수도 있다. 만약에 Port 와 Service 프로퍼티를 모두 지정한 경우라면 서버 소켓은 서비스 이름을 사용하게 된다.

- 클라이언트 요구 대기 (Listening for client request)

일단 서버 소켓 컴포넌트의 포트 번호를 설정하면, 런타임에서 Open 메소드를 사용하여 리스닝 연결(listening connection)을 생성할 수 있게 된다. 만약에 어플리케이션이 시작할 때 리스닝 연결을 자동으로 시작하게 하고 싶으면, 디자인 시에 Active 프로퍼티를 True 로 설정하면 된다.

- 클라이언트에 대한 연결 생성

리스닝 서버 소켓 컴포넌트는 클라이언트 연결 요구를 받는 족족 이를 허용한다. 이렇게 되면 OnClientConnect 이벤트가 발생하며, 적절한 처리를 이벤트 핸들러에서 해주면 된다.

- 연결에 대한 정보 얻기

일단 서버 소켓에서 리스닝 연결을 시작하면, 서버 윈도우 소켓 객체를 사용해서 연결에 대한 정보를 얻을 수 있게 된다. 서버 윈도우 소켓 객체는 Socket 프로퍼티를 이용해서 접근이 가능하다. 이 윈도우 소켓 객체를 이용하면 서버 소켓 컴포넌트가 받아들인 클라이언트 소켓 객체들과의 모든 활성화된 연결을 찾아낼 수 있으며, SocketHandle 프로퍼티를 이용해서 소켓 연결에 대한 핸들을 얻을 수도 있다. 이 핸들을 사용해서 윈도우 소켓 API 를 직접 호출할 수도 있다. 또한, Handle 프로퍼티를 이용하면 소켓 연결에서 날아온 메시지를 받은 윈도우에 접근할 수 있다.
각각의 클라이언트 어플리케이션에 대한 활성화된 연결은 서버 클라이언트 윈도우 소켓 객체(TServerClientWinSocket)에 캡슐화 되어 있다. 이들 모두에게 접근할 때에는 서버 윈도우 소켓 객체의 Connections 프로퍼티를 이용하면 된다. 서버 클라이언트 윈도우 소켓 객체에는 연결의 양쪽 종료점을 형성하는 클라이언트와 소켓 객체에서 사용하는 포트 번호

와 주소를 결정할 수 있는 프로퍼티가 있으며, SocketHandle 이라는 프로퍼티를 사용하면 윈도우 소켓 API 호출을 할 때 사용할 수 있는 소켓 연결에 대한 핸들을 얻을 수도 있다. 또한, Handle 프로퍼티를 사용하면 소켓 연결에서의 메시지를 받은 윈도우에 접근할 수 있으며, ASyncStyles 프로퍼티에서 메시지의 종류를 결정할 수도 있다.

- 연결의 종료

리스닝 연결을 종료할 때에는 Close 메소드를 사용한다. 이 메소드를 통해 클라이언트 어플리케이션과의 모든 연결을 단절시킬 수 있으며, 리스닝 연결이 종료 되므로 서버 소켓은 더 이상 새로운 연결을 허용하지 않게 된다.
클라이언트가 서버 소켓과의 연결을 종료하면 OnClientDisconnect 이벤트가 발생하며, 적절한 이벤트 핸들러를 이용해서 정리를 해준다.

소켓 이벤트

앞에서 몇 가지 이벤트에 대하여 이미 언급한 바가 있다. 이를 간단히 먼저 정리하면 논-블로킹 연결이나 블로킹 연결에서 소켓 연결에 대해 언제 데이터를 읽고, 쓸 것인지에 대한 이벤트가 있으며, 서버 측에서 연결을 종료할 때 클라이언트 소켓에서 받게 되는 OnDisconnect 이벤트, 그리고 클라이언트에서 연결을 종료할 때 서버 소켓에서 받게 되는 OnClientDisconnect 이벤트가 있다.
또한, 클라이언트와 서버 소켓 모두 연결에서 에러 메시지를 받게 되면 OnError 이벤트가 발생한다. 그 밖에 연결을 열고, 완료할 때까지 여러가지 이벤트가 발생하게 된다.

- 클라이언트 이벤트

클라이언트 소켓이 연결을 열게 되면, 다음과 같은 이벤트 들이 순차적으로 발생한다.

1. 서버 소켓을 찾기 전에 OnLookup 이벤트가 발생한다. 이 시점에서는 찾는 서버 소켓을 바꾸기 위해 Host, Address, Port, Service 프로퍼티를 변경할 수 없다. Socket 프로퍼티를 이용해서 클라이언트 윈도우 소켓 객체에 접근할 수 있으며, 그 객체의 SocketHandle 프로퍼티를 이용해서 윈도우 API 를 호출할 수 있다.
2. 윈도우 소켓이 설정되고, 초기화 된다.
3. 일단 서버 소켓을 찾으면 OnConnecting 이벤트가 발생한다. 이 시점에서는 윈도우 소
4. 켓 객체를 사용해서 서버 소켓에 대한 정보를 얻을 수 있게 되며, 연결에 사용되는 실제 포트와 IP 주소를 알 수 있다.
5. 연결 요구가 서버에 의해 받아들여지면 클라이언트 소켓에서 연결이 완료된다.

6. 연결이 완료되면 OnConnect 이벤트가 발생한다. 연결이 완료되는 즉시 데이터를 읽고, 쓰는 작업을 해야 할 때에는 OnConnect 이벤트 핸들러에 적절한 코드를 작성하면 된다.

- 서버 이벤트

서버 소켓 컴포넌트는 리스닝 연결과 클라이언트 연결의 두가지 형태의 연결을 형성할 수 있다. 이들 각각에 따라 발생하는 이벤트가 다르기 때문에, 따로 나누어 설명하겠다.

- 리스닝 연결 이벤트

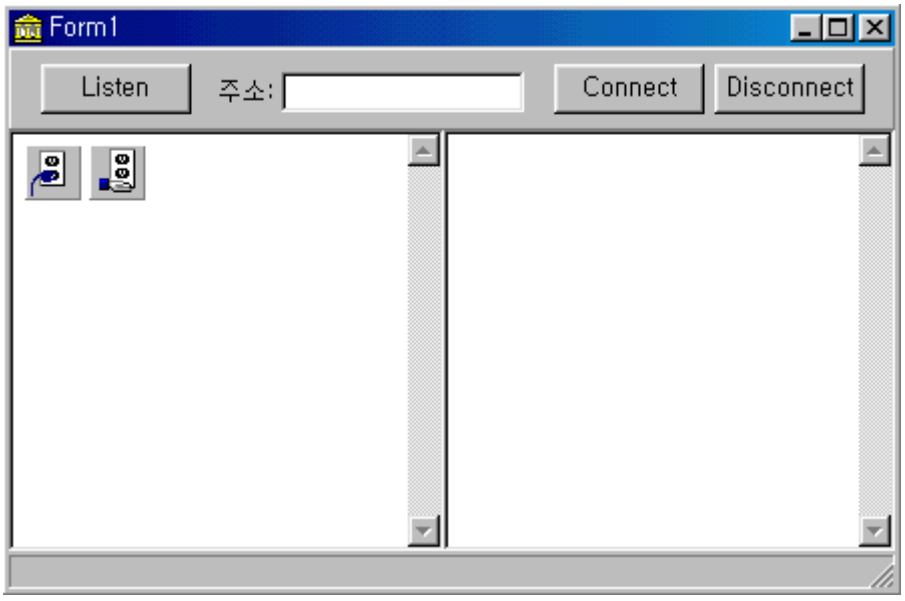
리스닝 연결이 생성되기 직전에 OnListen 이벤트가 먼저 발생한다. 이 시점에서 Socket 프로퍼티를 이용해서 서버 윈도우 소켓 객체에 접근할 수 있게 된다. 이 객체의 SocketHandle 프로퍼티를 이용해서 연결이 생성되기 전에 소켓에 대한 여러가지 사항을 바꾸어 볼 수가 있다. 예를 들어, 서버가 리스닝에 사용하는 IP 주소를 제한하는 등의 처리를 할 수가 있다.

- 클라이언트 연결 이벤트

1. 연결의 서버측 종료점을 형성하는 소켓에 대한 윈도우 소켓 핸들을 넘겨주는 OnGetSocket 이벤트가 클라이언트 측에 소켓 객체를 넘겨줄 때 발생한다. 이때 개발자가 TServerClientWinSocket 클래스 대신에 자신만의 윈도우 소켓 객체를 대신 사용하도록 할 수 있다. 즉, OnGetSocket 이벤트 핸들러에서 자신의 윈도우 소켓 객체를 생성할 수 있다.
2. 이어서 OnAccept 이벤트가 발생하는데, 여기에는 새로운 TServerClientWinSocket 객체를 이벤트 핸들러에 넘겨 준다. 여기에서 처음으로 TServerClientWinSocket 객체를 이용해서 클라이언트에 연결된 서버측 종료점에 대한 정보를 이용할 수 있게 된다.
3. ServerType 이 stThreadBlocking 이면 OnGetThread 이벤트가 발생한다. 여기에서 자신만의 TServerClientThread 객체를 생성할 수 있으며, 이를 대신 사용하게 된다. 이어서 쓰레드가 실행되기 시작하면 OnThreadStart 이벤트가 발생한다. 쓰레드의 초기화나 쓰레드가 연결을 통해 데이터를 읽고, 쓰는 작업을 하려고 윈도우 소켓 API 를 호출해야 한다면, 이 이벤트 핸들러에서 작업을 한다.
4. 클라이언트가 연결을 완료하면 OnClientConnect 이벤트가 발생한다. 논-블로킹 서버의 경우에는 이 시점에서 데이터를 읽고, 쓰기를 시작하면 된다.

1:1 채팅 예제의 제작

그러면, 실제로 예제를 만들어 나가면서 지금까지 설명한 것들을 익혀 보도록 하자.
이번에 만들 예제는 1:1 채팅을 가능하게 하는 프로그램으로 하나의 어플리케이션에 TClientSocket 과 TServerSocket 을 모두 올려 놓고, 이 프로그램이 경우에 따라서 채팅 서버가 되기도 하고, 클라이언트가 되기도 하는 프로그램이다.
본래 채팅 프로그램을 제대로 만들려면 서버 프로그램에 여러 개의 클라이언트가 접속하는 형태로 제작해야 하지만, 이 예제는 네트워크 프로그래밍의 기본을 이해시키려는 목적으로 제작하는 것이므로 1:1 채팅 만을 지원하도록 하였다.
이런 식으로 클라이언트와 서버의 기능을 모두 갖춘 프로그램은 프로그램을 테스트 하기에 편리하고 실제 메시지를 처리하면서 클라이언트와 서버에서 메시지를 처리하는 방법을 동시에 익힐 수 있는 장점이 있다.
먼저 폼에 메모 컴포넌트를 2 개와 TPanel, TStatusBar 컴포넌트를 하나씩 얹는다. 패널 위에는 버튼 컴포넌트 3 개와 IP 주소를 입력할 에디트 박스를 하나 추가하여 다음과 같이 디자인한다. 물론 1:1 채팅 프로그램을 만들기 위해서 TClientSocket 과 TServerSocket 컴포넌트도 추가해야할 것이다.



그리고 포트를 결정해야 하는데, 필자는 1001 번으로 결정하였다. ClientSocket1 과 ServerSocket1 컴포넌트의 Port 프로퍼티를 1001 로 설정하였다. 그리고, StatusBar1 컴포넌트를 선택하고 오른쪽 버튼을 클릭한 후 Panels Editor... 메뉴를 선택하여 패널 에디터를 띄운 뒤에 Add New(Ins) 버튼을 클릭하여 StatusPanel 을 하나 추가한다.
이제 이 1:1 채팅 어플리케이션은 클라이언트이면서 동시에 서버로 동작할 수 있도록 준비가 끝난 셈이다. 이를 위해서 전역 변수를 2 개 추가해야 하는데, 하나는 채팅 어플리케이

션이 클라이언트가 접속하기를 기다리는지 여부를 결정하는 Listening 변수와 현재 서버로 동작하고 있는지를 나타낼 IsServer 변수가 그것이다.

```
var
  Form1: TForm1;
  Listening: Boolean;
  IsServer: Boolean;
```

폼의 OnCreate, OnClose 이벤트 핸들러를 다음과 같이 작성하여 처음 폼이 생성될 때에는 Listening 변수를 초기화 하고, 폼을 닫을 때에는 소켓을 닫도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Listening := False;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ServerSocket1.Close;
  ClientSocket1.Close;
end;
```

Listen 버튼의 OnClick 이벤트에서는 현재의 폼을 서버로 대기하도록 하는 기능을 한다. 버튼의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Listening := not Listening;
  if Listening then
  begin
    ClientSocket1.Active := False;
    ServerSocket1.Active := True;
    Statusbar1.Panels[0].Text := 'Listening...';
  end
  else
  begin
```

```
    if ServerSocket1.Active then
      ServerSocket1.Active := False;
    Statusbar1.Panels[0].Text := '';
  end;
end;
```

이 버튼을 클릭하면 현재의 Listening 변수의 값을 변경한다. 그리고, 이 변수의 값이 True 이면 ServerSocket1 의 Active 프로퍼티를 True 로 설정하고 StatusPanel 의 Text 프로퍼티를 'Listening...'으로 설정한다. 이 값이 False 이면 서버 소켓의 Active 프로퍼티를 False 로 설정한다.
Connect 버튼을 클릭하면 Edit1 의 내용을 ClientSocket1 컴포넌트의 Address 프로퍼티로 설정한다. 참고로 앞에서도 설명했듯이 소켓의 Address 와 Host 프로퍼티 중에서 하나를 이용하는데, Address 프로퍼티는 IP 주소를 이용한다. 그리고, 이 주소를 이용하여 서버 소켓에 접속한다.
그리고, Disconnect 버튼을 클릭하면 클라이언트 소켓을 닫고, Listen 버튼의 OnClick 이벤트 핸들러를 호출한다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if ClientSocket1.Active then ClientSocket1.Active := False;
  if Length(Edit1.Text) > 0 then
  begin
    ClientSocket1.Address := Edit1.Text;
    ClientSocket1.Active := True;
  end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  ClientSocket1.Close;
  Button1Click(nil);
end;
```

이제 각 버튼의 OnClick 이벤트 핸들러는 모두 작성하였다. 이제부터 서버 소켓과 클라이언트 소켓의 이벤트 핸들러를 하나씩 작성하면서 이들의 역할을 알아보도록 하자.
먼저 서버 소켓의 이벤트 핸들러를 작성하도록 하자. 앞에서도 설명했듯이 서버 소켓에 클

라이언트 소켓이 접속을 시도하면 OnGetSocket 이벤트에 이어서 OnAccept 이벤트가 발생한다. 이 이벤트에서는 서버가 클라이언트의 접속을 받아들이기로 할 때 발생하는 이벤트이므로 IsServer 변수를 True 로 설정하여 이제 어플리케이션이 서버로 동작하고 있음을 나타내게 하고, StatusPanel 에 연결된 클라이언트의 IP 주소를 나타내도록 한다. OnAccept 이벤트의 Socket 파라미터는 클라이언트 소켓을 나타낸다.

```
procedure TForm1.ServerSocket1Accept(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  IsServer := True;
  Statusbar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteAddress;
end;
```

이어서 클라이언트가 연결을 완료하면 OnClientConnect 이벤트가 발생한다. 지금 작성하고 있는 채팅 어플리케이션과 같은 논-블로킹 서버의 경우에는 이 시점에서 데이터를 읽고, 쓰기를 시작할 수 있다. Memo2 컴포넌트에는 클라이언트 소켓에서 발생한 메시지를 표시할 것이므로, 이 시점에서부터 내용을 보여주도록 메모 컴포넌트의 내용을 지우도록 한다. 그리고, OnRead 이벤트는 서버 소켓이 클라이언트 소켓으로부터 데이터를 전달받을 때 발생하는데 Socket 파라미터의 ReceiveText 프로퍼티에 클라이언트 소켓에서 전송한 텍스트 값이 들어가 있다. 그러므로 이를 Memo2 컴포넌트에 보여주도록 이벤트 핸들러를 작성한다.

```
procedure TForm1.ServerSocket1ClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo2.Lines.Clear;
end;

procedure TForm1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo2.Lines.Add(Socket.ReceiveText);
end;
```

마지막으로 연결이 종료될 때에 발생하는 OnClientDisconnect 이벤트 핸들러에서는 서버 소켓의 Active 프로퍼티를 False 로 설정하고, 처음의 Listening 상태로 들어가기 위해서

Button1Click 이벤트 핸들러를 다시 호출한다. 이때 이를 호출하면 Listening 변수의 값이 변경되므로 먼저 Listening 변수 값을 변경한 뒤에 호출해야 원래의 값이 보존될 것이다.

```
procedure TForm1.ServerSocket1ClientDisconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  ServerSocket1.Active := False;
  Listening := not Listening;
  Button1Click(nil);
end;
```

이번에는 클라이언트 소켓의 이벤트 핸들러를 작성할 차례이다. 클라이언트 소켓도 서버 소켓과 접속이 되었을 때 OnConnect 이벤트가 발생한다. 여기에서도 마찬가지로 접속된 서버 소켓이 위치한 컴퓨터의 이름을 나타내도록 하는데, Socket 파라미터의 RemoteHost 프로퍼티를 사용하면 마이크로소프트의 UNC 이름에 해당되는 컴퓨터 이름이 나타난다.

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Statusbar1.Panels[0].Text := 'Connected to: ' + Socket.RemoteHost;
end;
```

이어서 나타날 수 있는 OnRead 이벤트 핸들러에서는 서버 소켓을 Socket 파라미터에서 얻을 수 있다. 서버의 텍스트 내용을 클라이언트로 동작하고 있는 어플리케이션의 메모 컴포넌트에 보여주도록 다음과 같이 이벤트 핸들러를 작성한다.

```
procedure TForm1.ClientSocket1Read(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo2.Lines.Add(Socket.ReceiveText);
end;
```

클라이언트 소켓의 OnDisconnect 이벤트 핸들러에서는 Button1Click 이벤트 핸들러를 호출하여 서버 소켓으로 동작할 수 있도록 한다.

```
procedure TForm1.ClientSocket1Disconnect(Sender: TObject;
```

```
  Socket: TCustomWinSocket);
begin
  Button1Click(nil);
end;
```

그리고, 이런 접속과정에서 에러가 발생할 경우에는 OnError 이벤트가 발생하는데, 이벤트 핸들러를 다음과 같이 작성하여 발생된 에러 상황을 나타내도록 한다.

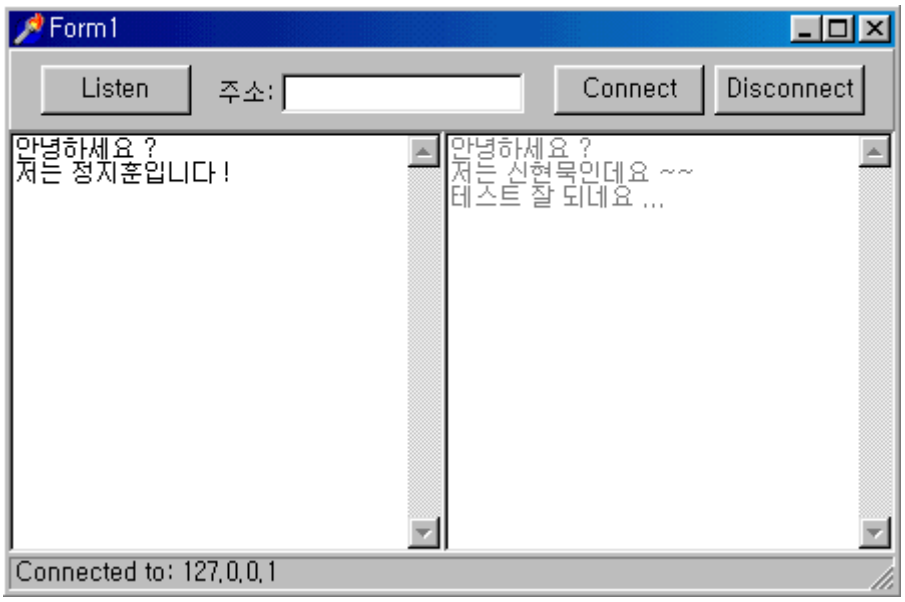
```
procedure TForm1.ClientSocket1Error(Sender: TObject;
  Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;
  var ErrorCode: Integer);
begin
  Memo2.Lines.Add('Error connecting to : ' + Edit1.Text);
  ErrorCode := 0;
end;
```

마지막으로 Memo1 컴포넌트의 OnKeyDown 이벤트 핸들러에서 눌려진 키가 리턴 키일 경우에 소켓으로 전송하도록 한다. 이때 IsServer 변수를 검사하여 서버일 경우와 클라이언트일 경우에 서로 다른 소켓에다가 SendText 메소드를 이용하여 텍스트를 전송한다.

```
procedure TForm1.Memo1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_Return then
    if IsServer then
      ServerSocket1.Socket.Connections[0].SendText(Memo1.Lines[Memo1.Lines.Count - 1])
    else
      ClientSocket1.Socket.SendText(Memo1.Lines[Memo1.Lines.Count - 1]);
end;
```

여기서 클라이언트 소켓의 경우에는 간단하지만, 서버 소켓의 경우 여러 개의 클라이언트 소켓과 물릴 수 있기 때문에 Connections 라는 컬렉션 객체가 포함되어 있다. 그렇지만, 우리가 작성한 어플리케이션은 단지 1:1 통신만 지원하므로 Connections[0]으로 접속된 클라이언트 소켓을 지칭할 수 있다.
이것으로 간단한 1:1 통신을 지원하는 채팅 어플리케이션이 완성되었다. 컴파일하고 실행한 뒤에 서버와 클라이언트 컴퓨터에 각각 띄우도록 한다.

그리고, 서버 측에서는 Listen 버튼을 클릭하여 클라이언트 프로그램의 접속을 대기하도록 하고, 클라이언트 측에서는 에디트 박스에 IP 주소를 적어 넣은 뒤에 Connect 버튼을 클릭하여 서버에 접속하도록 하자. 성공적으로 접속이 되면, 상태 바에 연결된 컴퓨터의 IP 주소(서버 컴퓨터의 경우) 또는 연결된 컴퓨터의 컴퓨터 이름(클라이언트 컴퓨터의 경우)이 나타날 것이다. 이제 메모 컴포넌트에서 통신을 시도하면 다음과 같이 채팅을 할 수 있을 것이다.



블로킹 연결을 이용한 파일 전송 예제

소켓을 이용한 프로그래밍을 할 때 앞서 설명한 채팅 어플리케이션과 같이 연결을 유지하면서 통신을 할 필요가 있을 때에는 중단되지 않는 논-블로킹 연결을 지원하도록 해야 하지만, 경우에 따라서는 전송하는 측과 전송받는 측의 데이터 전송에 있어서 서로 비동기 적으로 처리하는 것이 효율적일 때가 많다. 이럴 때에는 블로킹 연결을 이용하게 되는데, 블로킹 연결을 이용하여 소켓 프로그래밍을 하는 것은 해당되는 연결의 쓰레드를 생성하여 이를 실행하도록 하는 형태로 제작해야하기 때문에 논-블로킹 연결을 지원하는 어플리케이션에 비해 다소 까다로운 점이 많다.

그러면, 블로킹 연결을 통해 클라이언트에서 서버로 지정된 파일을 전송하는 예제를 작성해 보도록 하자. 이 예제는 Stig Johansen 이 공개한 예제를 바탕으로 작성하였다. 예제 프로그램의 구조는 클라이언트 어플리케이션에서 파일 열기 대화 상자에서 지정한 파일을 에디트 박스에 지정한 IP 주소로 전송한다. 그리고, 서버 어플리케이션에서는 클라이언트와 연결되면 서버에 저장할 파일 이름을 지정하면, 여기에 파일을 저장하도록 한다.

먼저, 서버 어플리케이션을 작성하도록 하자. 폼 위에 다음과 같이 메모 컴포넌트와 서버 소켓, 파일 저장 대화상자 컴포넌트를 하나씩 추가하고, 메모 컴포넌트의 Align 프로퍼티는

alClient, ScrollBars 프로퍼티는 ssVertical, ReadOnly 프로퍼티는 True 로 설정한다. 그리고 ServerSocket1 컴포넌트의 ServerType 프로퍼티는 stThreadBlocking, Port 프로퍼티는 2001 로 설정하도록 하자.

블로킹 연결을 지원하는 서버를 작성하기 위해서는 TServerClientThread 에서 상속받은 쓰레드 클래스를 이용하는 것이 핵심이다. 여기서는 파일과 소켓의 스트림을 내부적으로 사용하여 파일의 전송을 구현하기 때문에, private 섹션에 소켓 스트림과 파일 스트림 필드를 추가하고 외부에서 접근할 수 있도록 public 섹션에 프로퍼티로 선언하도록 한다.

```
TServerThread = class(TServerClientThread)
private
  FSocketStream: TWinSocketStream ;
  FFileStream: TFileStream;
protected
  procedure ClientExecute; override;
public
  property SocketStream: TWinSocketStream read FSocketStream write FSocketStream ;
  property FileStream: TFileStream read FFileStream write FFileStream ;
end;
```

쓰레드 클래스에서 꼭 오버라이드해서 구현해야 하는 메소드가 ClientExecute 로, 다중 쓰레드를 지원하는 경우에 Execute 메소드를 오버라이드하는 것과 같은 역할을 한다. 다중 쓰레드 프로그래밍에 대해서는 41 장에서 자세히 다루므로 이를 참고하기 바란다.
폼의 OnCreate 이벤트 핸들러에서 서버 소켓의 Active 프로퍼티를 True 로 설정하여 서버 소켓이 클라이언트 소켓과의 연결을 받아들일 수 있도록 이를 열어놓도록 한다.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ServerSocket1.Active := True;
end;
```

그리고 서버 소켓의 OnAccept, OnListen, OnThreadStart, OnThreadStop 이벤트 핸들러를 다음과 같이 작성하여 클라이언트 소켓과의 연결 상황을 메모 컴포넌트에 나타내도록 한다.

```
procedure TForm1.ServerSocket1Accept(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Accept ' + Socket.RemoteAddress);
end;

procedure TForm1.ServerSocket1Listen(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Listening ... ');
end;

procedure TForm1.ServerSocket1ThreadStart(Sender: TObject;
  Thread: TServerClientThread);
begin
  Memo1.Lines.Add('Start Thread of ' + Thread.ClientSocket.LocalAddress);
end;

procedure TForm1.ServerSocket1ThreadEnd(Sender: TObject;
  Thread: TServerClientThread);
begin
  Memo1.Lines.Add('End Thread');
end;
```

블로킹 연결에 있어서 가장 중요한 것은 OnGetThread 이벤트 핸들러에서 서버 쓰레드를 생성하는 작업이다. 서버 쓰레드를 생성할 때 두번째 파라미터를 False 로 선언하면 생성과 동시에 실행되는 것이지만, 다음과 같이 True 로 설정하면 쓰레드 객체의 프로퍼티를 변경한 뒤에 Resume 메소드로 쓰레드가 실행된다.

```
procedure TForm1.ServerSocket1GetThread(Sender: TObject;
  ClientSocket: TServerClientWinSocket;
  var SocketThread: TServerClientThread);
var
  SocketStream: TWinSocketStream ;
  FileStream: TFileStream ;
  FileName: string;
begin
  FileName := 'Default.file';
  SocketStream := TWinSocketStream.Create(ClientSocket, 20);
  if SaveDialog1.Execute then FileName := SaveDialog1.FileName;
  FileStream := TFileStream.Create(FileName, fmCreate or fmShareExclusive);
  SocketThread := TServerThread.Create(True, ClientSocket);
  (SocketThread as TServerThread).SocketStream := SocketStream;
  (SocketThread as TServerThread).FileStream := FileStream;
  SocketThread.FreeOnTerminate := True ;
  SocketThread.Resume ;
end;
```

여기에서 전송되어온 파일의 이름을 대화 상자에서 선택할 수 있도록 하고, 파일 스트림 객체를 생성하여 쓰레드 객체의 프로퍼티에 대입하고, 마찬가지로 TWinSocketStream 객체를 생성하여 이를 소켓 스트림 프로퍼티에 대입한다.
이제 쓰레드가 실제로 실행되는 ClientExecute 메소드를 구현하면 서버 프로그램이 완성된다. 이를 구현하기에 앞서 소켓 스트림의 전체 내용을 읽어오는 역할을 하는 ReadStream 함수를 다음과 같이 구현한다.

```
function ReadStream (Stream: TWinSocketStream; Buffer: Pointer;
  Count: Integer): Boolean;
var
  P: PChar;
  Total, Delta, TimeOut: Integer;
begin
  if Count = 0 then
  begin
    Result := True;
```

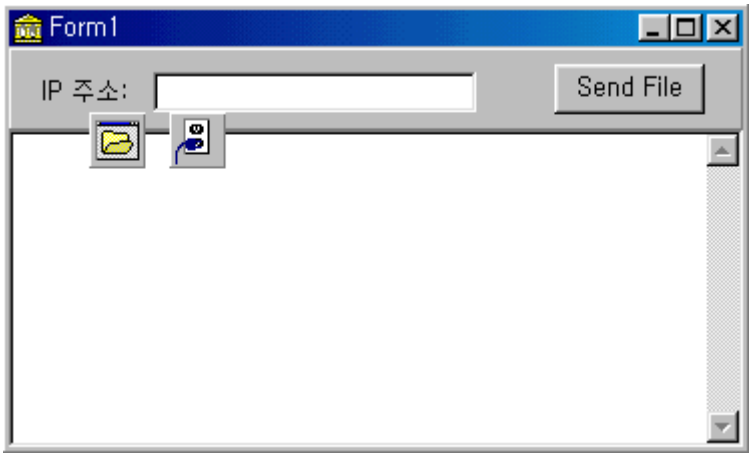
```
      Exit;
    end;
    TimeOut := 0;
    Result := True;
    Total := 0;
    P := Buffer;
    while Total < Count do
    begin
      try
        Delta := Stream.Read(P^, Count - Total);
      except
        Exit;
      end;
      if Delta = 0 then
      begin
        Inc(Timeout);
        while not Stream.WaitForData(1000) and (TimeOut < 20) do Inc(TimeOut);
        if Timeout >= 20 then
        begin
          Result := False;
          Exit;
        end;
      end
      else
        TimeOut := 0;
      Inc(P, Delta);
      Inc(Total, Delta);
    end;
  end;
```

이 루틴은 꽤 유용하게 사용되므로 잘 익혀두기 바란다. 구현된 내용을 간단히 설명하면 소켓 스트림에서 데이터를 읽어올 때 버퍼와 시간 제한을 이용하여 적절한 버퍼링을 해주는 것이 주된 내용이다. 이런 작업이 필요한 이유는 이런 버퍼링 작업이 없이 송신 측에서는 무조건 데이터를 밀어 넣고, 수신 측에서는 무조건 데이터를 가져올 경우에는 간혹 손실되는 패킷이 생기기 때문이다. 실제로 뉴스 그룹에서도 이런 문제로 어려움을 겪는 많은 개발자들이 있었는데, 이런 문제를 이 루틴으로 해결할 수 있다.

소스를 보면 쉽게 이해할 수 있을 것이나 간단한 설명을 덧붙이자면, 스트림의 Read 메소드에 의해 실제 읽어온 데이터의 바이트 수를 Delta 라는 변수에 대입하게 되는데 이때 이 값이 0 이면 WaitForData 메소드를 이용하여 버퍼에 데이터가 들어오는지 기다려 보고, 이를 여기서는 최대 20 번까지 기다려본 후에 그래도 데이터가 전송되어오지 않으면 연결이 끊어진 것으로 간주하고 실행을 중지하게 된다.
이 루틴을 이용하여 TServerThread 객체의 ClientExecute 메소드는 다음과 같이 구현한다.

```
procedure TServerThread.ClientExecute;
var
  FileLength: Integer;
  MemoryStream: TMemoryStream;
begin
  if ReadStream(SocketStream, Addr(FileLength), SizeOf(FileLength)) then
  begin
    MemoryStream := TMemoryStream.Create;
    MemoryStream.SetSize(FileLength);
    ReadStream(SocketStream, MemoryStream.Memory, FileLength);
    FFileStream.CopyFrom(MemoryStream, MemoryStream.Size);
    MemoryStream.Free;
  end;
  if Assigned(FSocketStream) then FSocketStream.Free;
  if Assigned(FFileStream) then FFileStream.Free;
  Terminate;
end;
```

여기서 눈여겨 보아야 할 것은 파일 스트림과 소켓 스트림의 원활한 전달을 위해 중간에 TMemoryStream 형의 변수를 이용한다는 점이다. 그리고, 처음에 일단 ReadStream 메소드를 이용하여 전달된 파일의 크기를 먼저 받아본다는 점이 중요하다. 이는 클라이언트에서 파일을 전송할 때 먼저 파일의 크기를 전송하고 나서, 실제 파일을 전송한다는 것을 의미한다. 그리고, 이 값을 이용하여 서버에서 파일을 생성하고 스트림을 복사한다.
이것으로 서버 프로그램이 완성되었다.
이번에는 이 서버 프로그램과 연결해서 사용할 클라이언트 프로그램을 작성해보자.
서버와는 달리 클라이언트 프로그램에는 연결한 서버의 IP 주소를 적어넣을 에디트 박스와 버튼을 하나의 패널에 올려 놓고, 메모 컴포넌트를 다음과 같이 추가하여 디자인하도록 하자.



클라이언트 소켓의 ClientType 프로퍼티는 ctBlocking 으로 설정하고, Port 프로퍼티는 서버와 같은 2001 로 설정한다.
그리고 먼저 클라이언트 소켓의 OnConnect, OnDisconnect, OnError 이벤트 핸들러를 다음과 같이 작성하여 통신 상황을 나타내도록 한다.

```
procedure TForm1.ClientSocket1Connect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Connected to ' + Socket.RemoteAddress);
end;

procedure TForm1.ClientSocket1Disconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  Memo1.Lines.Add('Disconnected to ' + Socket.RemoteAddress);
end;

procedure TForm1.ClientSocket1Error(Sender: TObject;
  Socket: TCustomWinSocket; ErrorEvent: TErrorEvent;
  var ErrorCode: Integer);
begin
  Memo1.Lines.Add('Error Code is ' + IntToStr(ErrorCode));
end;
```

그리고, 서버 프로그램에서의 ReadStream 과 마찬가지로 클라이언트에서 파일의 내용을 파

일 스트림에 읽어들인 후, 이를 소켓 스트림에 기록하는 역할을 하는 WriteStream 함수를 다음과 같이 구현한다.

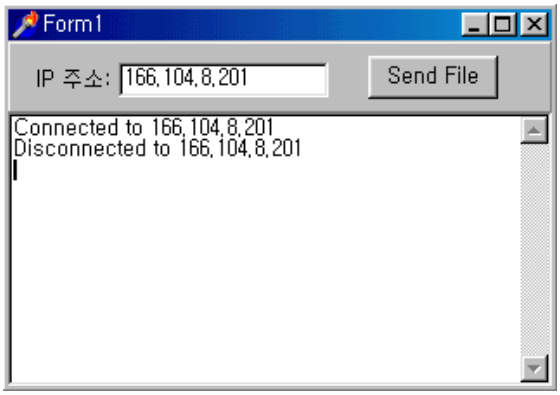
```
procedure WriteStream(Stream: TWinSocketStream; const Buffer: Pointer;
  Count: Integer);
var
  P: PChar;
  Total, Delta, TimeOut: Integer;
begin
  if Count = 0 then Exit;
  Total := 0;
  TimeOut := 0;
  Delta := 0;
  P := Buffer;
  while Total < Count do
  begin
    try
      Delta := Count - Total;
      if Delta > 16384 then Delta := 16384;       //최대값
        Delta := Stream.Write(P^, Delta);
    except
      Exit;
    end;
    Inc(P, Delta);
    Inc(Total, Delta);
  end;
end;
```

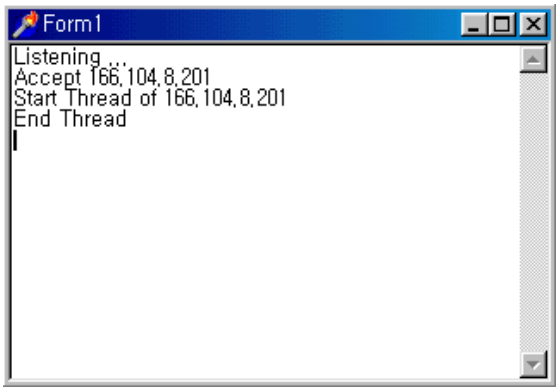
비교적 간단한 구현 내용이므로 쉽게 이해할 수 있을 것이다. 참고로 여기서는 한번에 최대 16384 바이트를 하나의 패킷으로 스트림에 기록하도록 하였다. 소켓 연결 상태 등에 따라 크기를 변경할 수도 있겠다.
마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStream: TFileStream;
```

```
  FileLength: Integer;
begin
  if OpenDialog1.Execute then
  begin
    FileStream := TFileStream.Create(OpenDialog1.FileName, fmOpenRead
      or fmShareDenyNone);
    FileLength := FileStream.Size;
    if FileLength > 0 then
    begin
      ClientSocket1.Address := Edit1.Text;
      ClientSocket1.Active := True;
      if ClientSocket1.Active then
      begin
        ClientSocket1.Socket.SendBuf(FileLength, SizeOf(FileLength));
        ClientSocket1.Socket.SendStream(FileStream);
      end;
        ClientSocket1.Active := False;
    end;
  end;
end;
```

일단 파일 열기 대화상자를 이용하여 전송할 파일을 선택하게 하고, 이 파일에 대한 파일 스트림 객체를 생성한다. 그리고, 파일 스트림의 크기를 먼저 SendBuf 메소드를 이용하여 전송하고, 파일 스트림의 내용을 SendStream 메소드를 이용하여 전송한다.
이와 같이 소켓을 이용하여 스트림을 전송할 때에는 패킷을 나누어 전송하고, 스트림의 크기를 먼저 전송하게 하는 것이 에러를 줄일 수 있는 요령이다. 물론, 독자적인 프로토콜을 정의하여 이를 이용하는 것이 가장 이상적일 것이다.
이것으로 클라이언트 어플리케이션이 완성되었다. 이제 클라이언트와 서버 어플리케이션을 띄우고 파일을 선택하여 전송하도록 해보자. 다음 그림은 서버와 클라이언트 어플리케이션의 실행화면이다.





정 리 (Summary)

이번 장에서는 델파이에서 기본적으로 제공되는 소켓 컴포넌트를 이용하여 프로그래밍을 하는 방법에 대해서 알아보았다. 소켓 컴포넌트는 과거 뉴스그룹에서 델파이 판매 전략에서 C/S 버전에만 포함된 것이 격렬한 논란거리가 된 컴포넌트이다. 그렇기 때문에, 윈속을 지원하는 수많은 프리웨어 컴포넌트 들이 나오게 되었고, 델파이 4 에서는 프로페셔널 버전에 소켓 컴포넌트만 추가한 새로운 제품군을 등장시켰다. 그만큼 사용 빈도도 높고, 또한 쓸모도 많다.

그러므로, 여기에서 소개한 소켓 컴포넌트 뿐만 아니라, Francois Piette 가 공개한 프리웨어 컴포넌트 들도 매우 뛰어나고 더 잘된 것들도 많으므로 이런 프리웨어에도 관심을 두고 찾아보기 바란다.