How to use *TmsListenerSocket* component and *TmsServerThread* class

Use *TmsListenerSocket* component to develop server applications. It works in conjunction with *TmsServerThread* class, which is the descendant of Delphi *TThread* class, and controls the actual behaviour of the server you want to create. You should white this class yourself, and assign it to the *ServerThreadClass* property of *TmsListenerSocket* component.

Let's try to create very simple server application. It will listen on the port 1090 and, as soon as client connects to it, sends back a content of the file named *myfile.txt*.

Start Delphi IDE, and click File – New Application. On the *Form1*, drop *TmsListenerSocket* component, then, in the Object Inspector, set the *Port* property to 1090.

Now, let's write the descendant of *TmsServerThread* class, which will implement the actual behaviour of the server. We will have to create the class, which descends from *TmsServerThread*, and override the *execute* method.

Switch to the Unit1.pas by pressing F12, and, just above of the TForm1 class declaration, insert the following:

```
TMyServerThread = class(TmsServerThread)
protected
  procedure Execute; override;
end;
```

And, in the implementation section, type the following code:

```
procedure TmyServerThread.Execute;
var
  OutStream: TStream;
begin
  OutStream:=TFileStream.Create('myfile.txt',fmShareDenyWrite);
  try
    ServerSocket.SendStream(OutStream);
    ServerSocket.Disconnect;
  finally
    OutStream.Free;
  end;
end;
```

Note the usage of *fmShareDenyWrite* file open mode constant in the constructor of *OutStream*. If we use just *fdOpenWrite*, the program will crash if the server receives more than one request at the same time.

Now, let's go back to our form, and assign the OnCreate event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  msListenerSocket1.ServerThreadClass:=TMyServerThread;
  msListenerSocket1.Start;
end;
```

Here we are telling to the instance of TmsListenerSocket component, which code to use when a connection request arrives from the client, and also, starting up the server.

Please note, that we are assigning the type of the server code we just wrote, not the instance of TMyServerThread class… We never create an instance of our server classes. TmsListenerSocket component does it for us when it accepts the connection.

In the OnClose event handler we will have to stop the server:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  msListenerSocket1.Stop;
end;
```

Now, let's run the program, and try to connect to it using Telnet application, which comes with Windows. Run Telnet and click Connect – Remote System, then in the Host Name box type either localhost, or 127.0.0.1, and in the Port box – 1090 (the number of the port our server is listening), and click connect. You should see the content of myfile.txt, returned by our server.

As you can see, it is very simple to create the server application using these two classes. But now let's make the server more complex.

First, let's try to add logging of received connections.

Drop the TMemo component on the form, and then add the method UpdateStatus to the TmyServerThread class definition:

```
TMyServerThread = class(TmsServerThread)
private
  FStatusStr: string;
  procedure UpdateStatus;
protected
  procedure Execute; override;
end;
```

We also added FStatusStr field, which will be used to pass the status string to the UpdateStatus method, which will be called through TThread.Synchronize method.

In the implementation section, add the following code:

```
procedure TMyServerThread.UpdateStatus;
begin
  Form1.Memo1.Lines.Add(FStatusStr);
end;
```

And modify the TMyServerThread.Execute method so that it looks like the following:

```
procedure TmyServerThread.Execute;
var
  OutStream: TStream;
begin
  FStatusStr:='Reqested connection from '+Peer;
  Synchronize(UpdateStatus);
  OutStream:=TFileStream.Create('myfile.txt',fmShareDenyWrite);
  try
    FStatusStr:='Sending data to '+Peer;
    Synchronize(UpdateStatus);
    ServerSocket.SendStream(OutStream);
    FStatusStr:='Closing connection with '+Peer;
    Synchronize(UpdateStatus);
    ServerSocket.Disconnect;
  finally
    OutStream.Free;
  end;
end;
```

Now, run the program and connect to it using Telnet application, which comes with Windows… You will see a log of connection in Memo1.

Now, let's make the server thread even more complicated. In some cases you will need to perform certain initialisation and cleanup for server threads. For example, if you wish to use some local variables inside of threads, such as other VCL components, or allocate the memory for certain variables. Since you are not creating the instances of TmsServerThread descendants, you will have to use OnServerThreadStart and OnServerThreadTerminate event handlers of TmsListenerSocket component.

Let's modify our existing server so that it OutStream is declared as a property of TMyServerThread class, not as a local variable of TMyServerThread.Execute method.

So, now, the declaration of TMyServerThread class looks like this:

```
TMyServerThread = class(TmsServerThread)
private
  FOutStream: TStream;
  FStatusStr: string;
  procedure UpdateStatus;
protected
  procedure Execute; override;
public
  property OutStream: TStream read FOutStream write FOutStream;
end;
```

Now, TMyServerThread.Execute method will look like this:

```
procedure TmyServerThread.Execute;
begin
  FStatusStr:='Reqested connection from '+Peer;
  Synchronize(UpdateStatus);
  FStatusStr:='Sending data to '+Peer;
  Synchronize(UpdateStatus);
  ServerSocket.SendStream(FOutStream);
  FStatusStr:='Closing connection with '+Peer;
  Synchronize(UpdateStatus);
  ServerSocket.Disconnect;
end;
```

Now, we have to create OnServerThreadStart and OnServerThreadTerminate event handlers for TmsListenerSocket component. Click the msListenerSocket1 on the form, then go to the Object Inspector and assign above methods.

```
procedure TForm1.msListenerSocket1ServerThreadStart(Sender: TObject;
  ServerThread: TmsServerThread);
var
  TempStream: TStream;
begin
```

```
  TempStream:=TFileStream.Create('myfile.txt',fmShareDenyWrite);
  (ServerThread as TMyServerThread).OutStream:=TempStream;
end;
```

In this event handler we are creating the TFileStream object, and assigning it to the OutStream property of the instance of TMyServerThread, which has been created by msListenerSocket1, when it received a connection request.

In the OnServerThreadTerminate event handler we have to dispose the stream we created above:

```
procedure TForm1.msListenerSocket1ServerThreadTerminate(Sender:
TObject;
  ServerThread: TmsServerThread);
begin
  (ServerThread as TMyServerThread).OutStream.Free;
end;
```

This version does not do anything different from the previous one; it just illustrates how to initialise and cleanup the server thread.

Now, we should take special care of exception handling in the server thread. If an exception raises inside of the thread, we will have to pass it's handling to the main thread, otherwise our program may crash if there is even single exception in a single connection.

The simplest way of doing it is to call *Application.HandleException* method, which will display traditional dialog box with the standard exception message. But, it will mean that the server will require the user intervention in order to close this dialog box. So, what we are going to do is – just record the information about the exception into the log.

To do it, let's change the procedure TMyServerThread.Execute to the following:

```
procedure TmyServerThread.Execute;
begin
  FStatusStr:='Reqested connection from '+Peer;
  Synchronize(UpdateStatus);
  FStatusStr:='Sending data to '+Peer;
  Synchronize(UpdateStatus);
  try
    ServerSocket.SendStream(FOutStream);
  except
    on E:Exception do
    begin
      FStatusStr:='Error '+E.Message;
      Synchronize(UpdateStatus);
    end;
  end;
  FStatusStr:='Closing connection with '+Peer;
  Synchronize(UpdateStatus);
  ServerSocket.Disconnect;
end;
```

As you can see, the creation of complex server applications, using the components included in IMS is very simple.

The code of the application we just created is in the project ls1.dpr, and is included in the IMS package.