Development of Visual Delphi Components with Internet Access

By Roger Connell, Innova Solutions Pty Ltd.

Abstract

Any Internet development seems to require threads to operate correctly. Components within Delphi are a useful way of achieving code reuse. The paper looks at the process undergone to emulate the VCL components TDirectoryListBox and TFileListBox as WEB enabled components by encapsulating an Indy FTP Client in a separate thread. It looks at the development decisions, problems encountered and solutions implemented.

The following are discussed:
- The issues of dealing with Indy components in the VCL main thread.
- Splitting Component functionality between threads.
  - Tasking the Internet Thread.
  - Updating the VCL presentation with information returned.
  - Returning information to VCL main thread functions.

Introduction

This paper discusses the development of two Delphi components, a Network Directory List Box and a Network File List Box, which use an Indy FTP Client component to interact with a remote server using FTP. Two implementations are described. The first assumed that, with the small file sizes of the directory data, the FTP transfers could exist in the main thread. The second implementation uses an additional thread encapsulated in the Directory List components to resolve problems in the initial implementation.

The network components were required as part of a File Transfer Application development.

Why Develop a New FTP Client

I feel firstly I must justify in part why I chose to develop rather than buy an existing FTP Client.
- I wanted to do a serious Indy development. My other applications have used Netmasters components and I wanted to compare the Indy ones in a real project.
- It seemed very easy.
- There is a requirement for a batch upload/download FTP ability.
- I prefer long transfers to happen in the background.
- There is a requirement to easily launch repeated transfer actions.
- I wanted the ability to carry out multiple simultaneous transfers.

Quick Review of TCP/IP

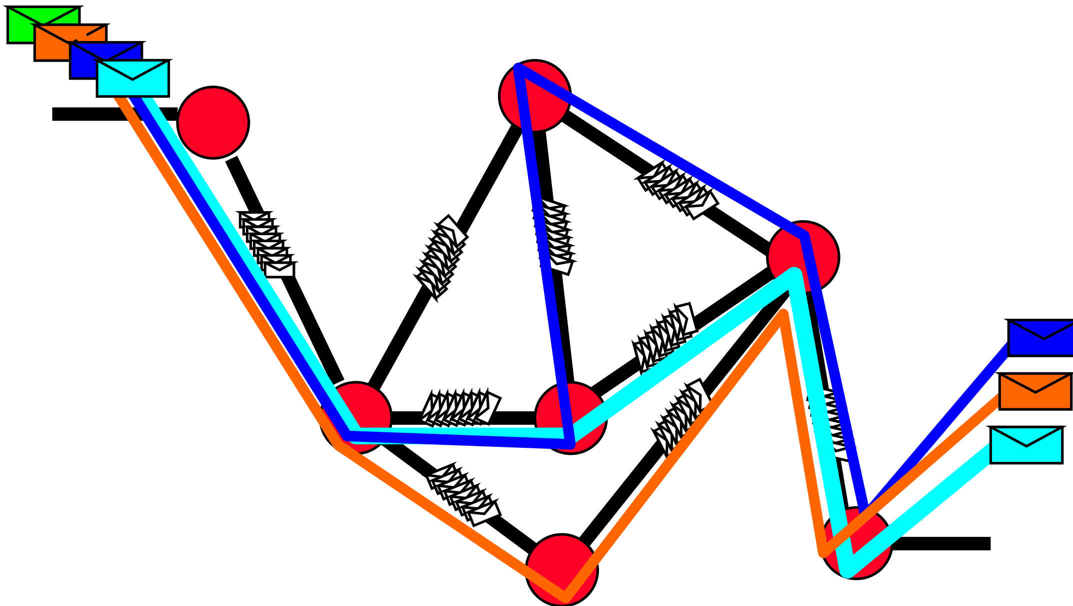
IP - Internet Protocol

IP or Internet Protocol forms the base on which the internet is built.

Before passing data to the IP Layer it must be fragmented into small, manageable blocks. In the IP layer each block is wrapped in an "envelope" or data packet with an "IP" address on it and released to the network. Each packet is analysed and checked at each network node. If it is damaged it is disposed of, otherwise it is passed on to the next node which appears closest to the destination address.

On each link in the system a packet will queue behind other packets waiting for that resource. As queues get longer the packet delay is increased. In the event of a link failure or severe congestion packets which follow will be routed by a better path.

Packet transmission time and reliability of delivery depend on network quality and traffic load.

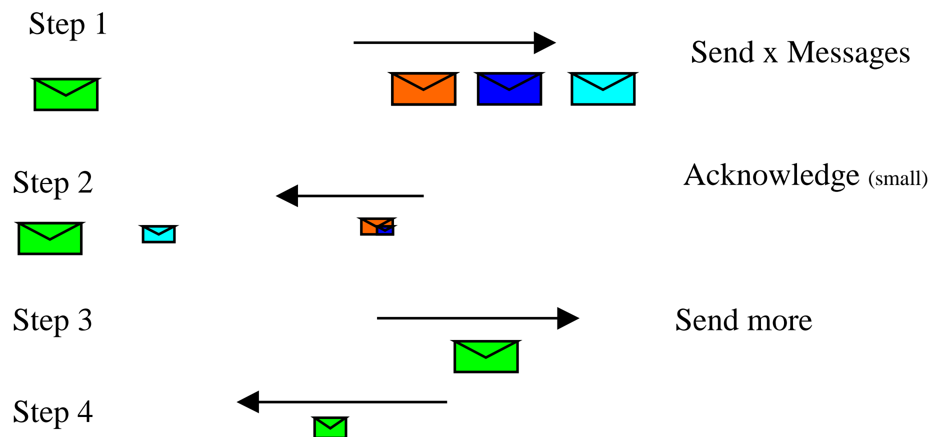


Pictorial representation of IP packet transmission in a network of IP Nodes

TCP – Transport Connection Protocol

TCP or Transport Connection Protocol establishes a transport layer which does the fragmentation and interacts with the IP Layer to confirm delivery of each fragment of data. It then reassembles the received data in the correct order.

Received data is acknowledged by sending small messages back to the sender. If data is not acknowledged it is resent after a timeout. A small number of IP messages are sent initially then others are sent as previous messages are acknowledged.



TCP Handshaking – Green packet is held until first packet is acknowledged

Network Implications

When parts of a network are busy the response times can become quite significant. Bottlenecks can occur if a host internet connection is very busy or part of the network is congested. There are many points of failure or inordinate delays which are beyond the control of client software.

A single FTP session may only use a portion of the available bandwidth at the client end, especially with broadband internet connections.

To make maximum use of broadband capability you need to be able to carry out multiple simultaneous transfers. Often the transfer rate is throttled by the access to the server or on the pipes connecting the service provider to the backbone network. Multiple sessions allow simultaneous transfer to a number of servers. Even when the congestion is in shared pipes to the backbone network, each session gets its own share of the bandwidth and, while the individual session transfer rate is slower, the total throughput is increased

The Initial Component Development Plan

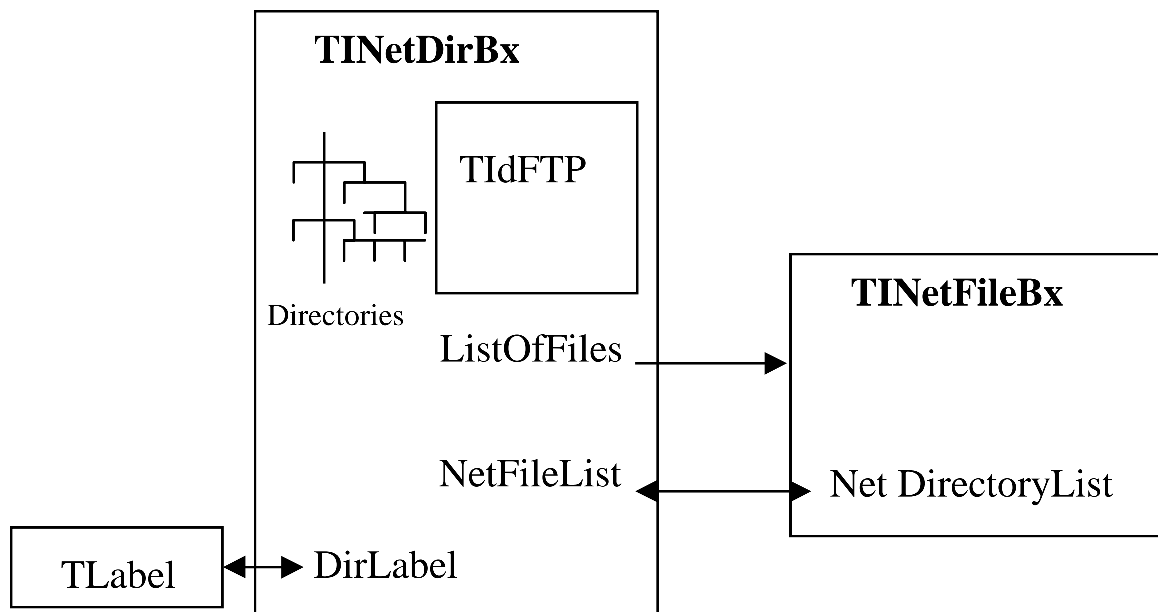
Having decided to develop yet another FTP Client with a Windows “look and feel”, I needed a visual directory listing to browse server directories and support “drag and drop” functions. This is ideal for component development and we would then have it for future applications.

It seemed so simple, inherit from TFileListBox and TDirectoryListBox, encapsulate an FTP Client and modify a couple of routines.

Further investigation showed that the directory access functions were deeply embedded in functions that revolved around filename manipulations to produce the required presentation. With the availability of source code for the VCL components it was still feasible to go back a level and inherit from their common base object, TCustomlistBox.

It seemed logical that the network interaction should be contained in the directory component so a TINetDirBx was produced which contains an FTP Client and a few properties were exported to allow it to be set up. Next, the published and public definitions were copied from the standard components. The standard Windows component implementations were then analysed and new method implementations produced. Many of the properties required simply expose existing TCustomlistBox functions.

It was also necessary to add some new functionality to map and navigate the Web based server. Other functions were added to enable the main application and any coupled TINetFileBx to access the server though the TINetDirBx component.



Initial Component Block Diagram

Problems Encountered in Mark 1

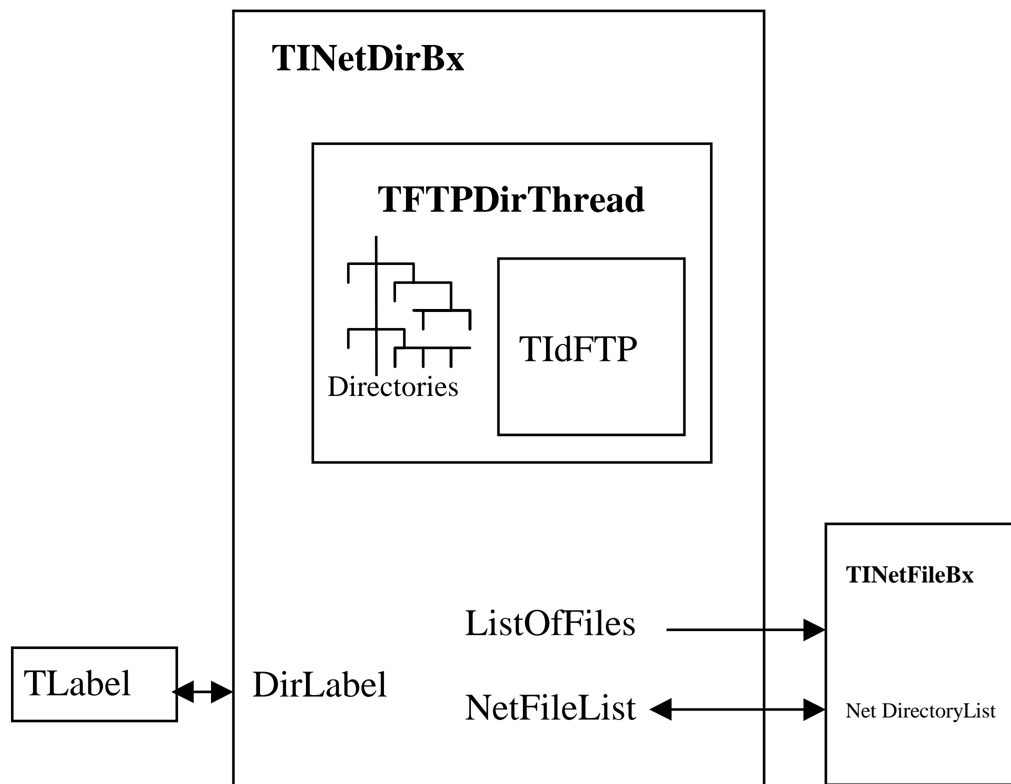
In the initial implementation the FTP Client operated within the VCL thread and it was thought that because the file transfers required are so small the effect on the thread would be tolerable. Most of the time this was the case but, if the remote server became busy or the access line was congested (e.g. with a previously initiated file transfer), then the stalling of the VCL thread was a little irritating.

Less tolerable was the occasional hanging of the component. This did not happen often but if the server terminated badly or stalled, or the dial up connection was killed, then the

FTP client would hang indefinitely and stop the VCL thread requiring that the application be forcibly terminated.

Introduce a Component Thread

The solution appeared to be encapsulating the FTP Client with its long waits and hang-ups in a Thread Object. Some time was spent considering the implications of this. Threads add an overhead but, given that the user understands that each TINetDirBx component contains a thread and does not use them lightly, there was no reason why a thread could not be encapsulated in a component to carry out the internet transactions.



Modified Component Block Diagram

The network status routines were then identified and moved into the new thread object and a method of communicating between the VCL and the new thread routines established.

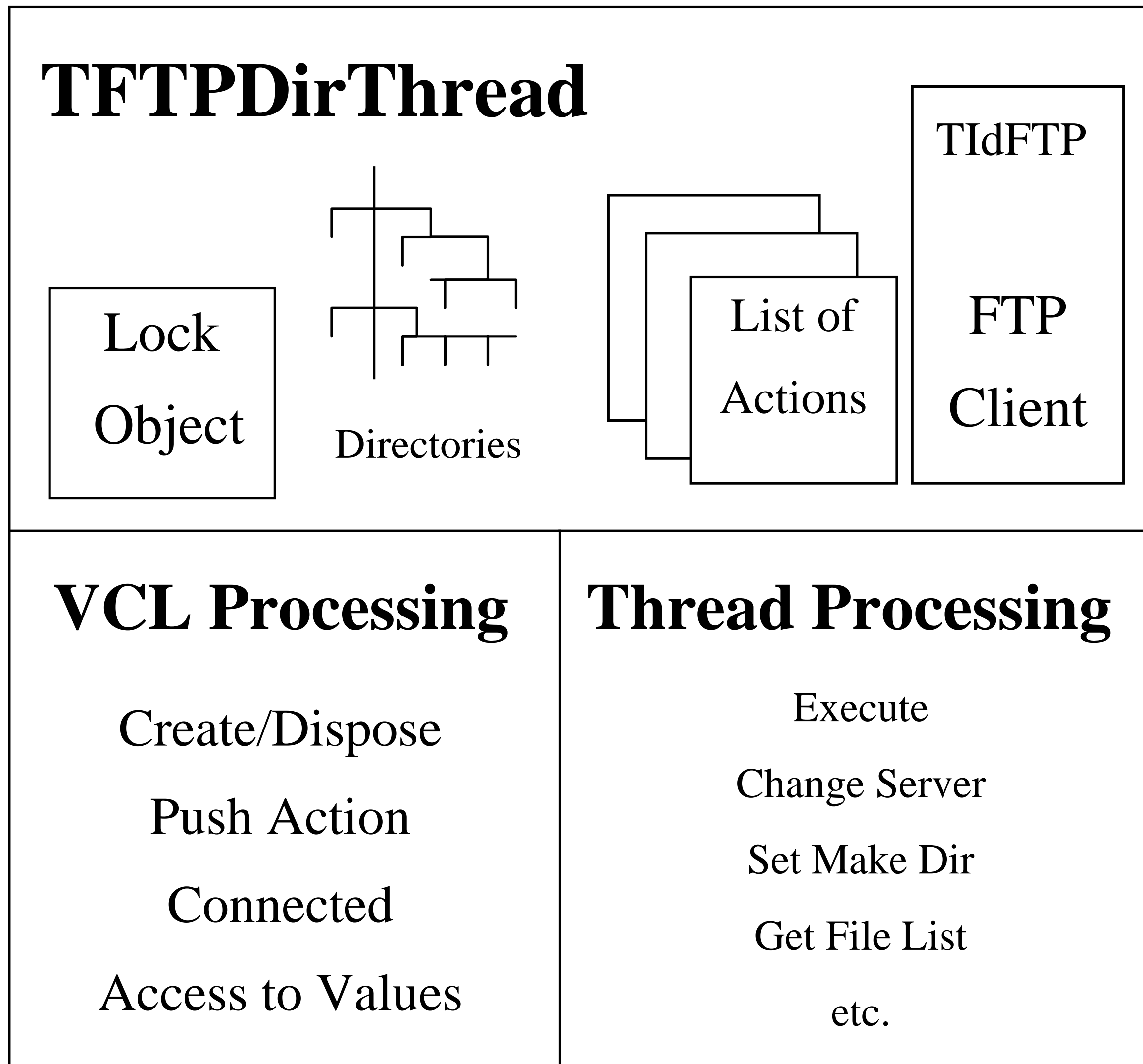
The thread object contains variables which are used by the routines within that object as expected, however some routines are “thread” routines, some are “VCL” routines and some are “synchronised” routines. If a variable is shared between a thread routine and a VCL routine the thread object arbitrates using a local TCriticalSection object. No arbitration is required for synchronised routines.

Review of Threads

A simple (single thread) computer program steps through instructions in sequence. If it needs to read data from a disk or the Internet the stepping stops until the read action completes.

The Windows operating system manages a number of "threads" each of which is able to step through separate sets of program instructions. Computer time is shared between the "threads" so that for x milliseconds one program executes then the next program executes for x milliseconds. If one program needs to wait for data its timeslot is terminated and control is passed to the next thread. In this way Windows can run a number of programs "at the same time".

The Operating System also permits multiple threads in the one program. The TThead Object in Delphi wraps this for our use. Having created a descendant of TThread the Delphi programmer gets a second execution thread to step through parts of the code "at the same time" as the main application actions are occurring. The new thread is controlled by overriding TThread.Execute. Execute is entered when the thread object is started. The additional thread terminates only after exiting Execute.



TFTPDirThread Object Construction

Thread Execution

The main application already used threads to manage file transfers and the same method of tasking the thread by special “Action” objects was used. In this instance it was a little more complicated as there were a whole range of functions instead of just “send” or “receive” a file.

The “Execute” routine in the thread object takes an action object off a list and calls the appropriate thread routine(s) to process that action. After completion it will then either process the next action on the list or go into a state of suspension.

Exceptions within threads always need to be handled as they will otherwise terminate the thread. Exceptions must be expected within any communications environment as networks or other remote events fail or abort a communications path. Exception handling in this case clears out any outstanding actions and executes a long timeout to give the network time to recover.

```
procedure TFTPDirThread.Execute;
begin
… … …
   While not Terminated do
      Begin
         If FFTPClient=nil then Terminate;
         If Terminated then Exit;
         FCurAction:=PopList;
         If Terminated then Exit;
         If FCurAction=nil then Suspend
         else
            Try
               Case FCurAction.Action of
                    taChangeServer:ChangeServer;
                    taDeleteF:DeleteSingleFile(FCurAction.CommandAction);
                    … … … etc … … …
               end;       //Case;
               If Terminated then Exit;
            Except
            on e:exception do
               Begin
                    If Terminated then Exit;
                    ReportFTPErrors(e.Message);
                    FreeAndNil(FCommandList);
                    Sleep(1000); // Give it time to get over it
               end;
            end;
         FreeAndNil(FCurAction);
      end;
end;
```

Launch a Task

Function stubs in the main directory object launch a thread action (e.g. LaunchRename). To do this they must create and populate an "Action" object and "Push" it into the thread object. In this case the "Action" object defines:

- Type of action,
- A command string, a string list pointer and two integers,
- A Return Function Pointer.

The type of action determines the function to be called in the Thread.

The command string is used to pass string parameters such as filenames into the thread routine.

Some thread routines require more parameters such as the list of files to delete or depth of directory search, hence the string list pointer and integers.

The return function pointer is used when the completion of the thread task requires more than updating the status and redrawing of the VCL components.

```
procedure TINetDirBx.LaunchRename( ACurrentFileName,
                                    ANewFileName: String);
Var
   NewAction:TFTPCompActionList;
begin
   If FDirFTPThread=nil then exit;
   NewAction:=TFTPCompActionList.Create;
   NewAction.Action:= taRename;
   NewAction.CommandAction:=  ACurrentFileName+
                              FileNameSeparater+ANewFileName;
   FDirFTPThread.PushList(NewAction);
end;
```

"Push" a Task onto Thread Task List

The first requirement for synchronisation occurs when attempting to put a task onto the thread action list as both threads will be required to write to the list location, one to add a task and one to remove it. The lock object is used make sure that only one thread is operating on this shared data at a time. "PushList" first waits until it can acquire the lock object. If the new action is a "change server" action it will then clear out any waiting tasks. The new task is then added to the end of the list of waiting tasks. The FTP thread is then "Resumed" in case it is in a suspended state. Finally the lock is released.

"PushList" is called by the VCL thread and the FTP thread will not be affected unless it happens to try and get a new task at this time in which case it will have to wait.

```
procedure TFTPDirThread.PushList( Item:TFTPCompActionList );
begin
    If FObjLock=nil then exit;
    If Item=nil then Exit;
    Try
        FObjLock.Acquire;
        If Item.Action = taChangeServer then
                FreeAndNil(FCommandList);
        If FCommandList=nil then
                FCommandList:=Item
        else
                FCommandList.AddToList(Item);
        If Suspended then Resume;
    Finally
        FObjLock.Release;
    end;
end;
```

"POP" a Task from the Thread Task List

The FTP Thread Execute routine uses "PopList" to get each new task after it has finished the previous task. PopList first waits to acquire a lock. It can then remove the top item from the list and release the lock.

```
function TFTPDirThread.PopList: TFTPCompActionList;
begin
    Result:=nil;
    If FObjLock=nil then exit;
    Try
        FObjLock.Acquire;
        If FCommandList=nil then exit;
        Result:= FCommandList;
        FCommandList:=Result.Next;
        Result.Next:=nil;
    Finally
        FObjLock.Release;
    end;
end;
```

Synchronize Function Call

Synchronize is the Delphi suggested method of updating VCL components with information created by a non VCL (main) thread. Synchronize requires as a parameter a TThreadMethod which is a procedure of object with no parameters. Use of Synchronize will result in this method being executed in the VCL thread. The calling thread is suspended while this happens.

Synchronize first holds the calling thread then posts a message to the main thread. In responding to this message, the main thread retrieves the appropriate TThreadMethod

and executes it before releasing the calling thread. The calling thread will raise an exception at this point if the call produced an exception in the VCL thread.

The TINetDirListBx Component implements two general synchronisation methods to return data from the FTP Client Thread to the main thread functions.

- A Standard Return
- A Special Return

Standard Return -

Most of the FTP Thread tasks or actions use the same routines to interact with the VCL. This interaction is contained in two TThreadMethods. ChangeDirectory is an example of a routine which uses only the standard return. The standard return does no more than manage the visual presentation of the components on the form. The standard implementation uses BusyCursorAndDisable at the start of the task and RecoverValues at the task completion.

Synchronize (BusyCursorAndDisable)

BusyCursorAndDisable is called at the start of those execute routines which will change the visual presentation of the component. It is synchronised with the VCL thread and flags components **TINetDirBx** and **TINetFileBx** as disabled. It also counts re-entry so that if the component is already flagged no action is taken. These execute routines can be called directly by a task or indirectly by another task via its primary routine. It is not desirable to redraw the component on each separate call.

On reflection it also does not make sense to synchronise for the update count only. This is a left over from the development path and will be looked at in the future.

Synchronize (RecoverValues)

RecoverValues is called at the completion of execute routines as they complete their tasks. On the last exit the call count drops to zero and the VCL components are redrawn with the new information gathered for that task by the FTP thread. Some remote server and network values are copied to the main VCL Objects.

A belts and braces call to RecoverValues is put in the main execute function to ensure exceptions do not upset the re-entry counter.

Special Return

For some tasks the thread task return needs to provide data to the main thread and may not even need to redraw the VCL components. FileExists is one such task, it is provided as a public service to the application form the component resides on and should not affect the visual presentation.

In this case the main thread call needs to appear as a normal method of TINetDirBox returning with a result of true or false. The TINetDirBx.FileExists call passes a taExistF action task to the FTPThread and then goes into a ProcessMessages loop.

The “Action” item is populated with a VCL return procedure.

At the completion of the taExistF task the thread execute calls Synchronize (ReturnProcedure). The thread object procedure ReturnProcedure is now executed in the VCL threads ProcessMessages loop. This procedure in turn calls the TINetDirBx.FileExistsReturn procedure which was passed via the "Action" item. This call has the threads FResultList passed as a parameter which can be analysed to determine if the file exists. TINetDirBx.FileExistsReturn then releases TINetDirBx.FileExists from its loop and it is able to report back the result.

The process messages loop currently waits (10 secs) before returning a "don't know" response. The VCL thread still operates as other messages are processed.

```
function TINetDirBx.FileExists(AFileName: string): TYesNoDontKnow;
Var
   NewAction:TFTPCompActionList;
   TryCount:Integer;
begin
   Result:=ynDontKnow;
   If FDirFTPThread=nil then exit;
   If FFileExistBusy then Exit;
   Try
         FFileExistBusy:=true;
         NewAction:=TFTPCompActionList.Create;
         NewAction.Action:= taExistF;
         NewAction.CommandAction:= AFileName;
         NewAction.ReturnProc:= FileExistsReturn;
         FFileExistingCompleted:=false;
         FFileExisting:=AFileName;
         FDirFTPThread.PushList(NewAction);
         TryCount:=0;
         While Not FFileExistingCompleted do
            Begin
               Sleep(500);
               Application.ProcessMessages;
               TryCount:=TryCount+1;
               If TryCount>20 then exit;//only wait 10 Seconds
            end;
         If FFileExisting='' then   result:=ynNo
            else If FFileExisting=AFileName then result:=ynYes;
   Finally
      FFileExistBusy:=False;
   end;
end;

procedure TFTPDirThread.ReturnProcedure;
begin
  If Assigned(FCurAction.ReturnProc) then FCurAction.ReturnProc(FResultList);
end;
```

```
procedure TINetDirBx.FileExistsReturn(ReturnList: TStringList); //Synchronized
begin
  If ReturnList=nil then FFileExisting:=''
    else
    Case ReturnList.Count of
        2:if pos(ReturnList[0],FFileExisting)<>0 then // this is the right file
            Begin
              if ReturnList[1]=ReturnValueNullResult then
                FFileExisting:='';
            end
          else   FFileExisting:=ReturnValueNullResult
        else //Case
          FFileExisting:=ReturnValueNullResult;
      end;//case
    FFileExistingCompleted:=true;
end;
```

Freeing The Components

Closing the application is an issue. If a thread in the application does not terminate then the application will not completely terminate. When the thread object is freed it sets Terminated and waits until the thread execute exits. If long waits occur in the thread it does delay the termination of the application.

All Object encapsulated in the components have to be freed so the TFTPDirThread.Destroy function acquires the lock before freeing the waiting action list. The FTPClient is aborted in the TFTPDirThread.Destroy before it is freed so that it does not leave some remote peer in an indeterminate state but terminates gracefully.

Conclusion

I hope that I have raised your awareness of the impact of using the main application thread when carrying out internet interaction and then demonstrated how threads avoid these impacts. The full source code for these components and the application executable is available at http://www.innovasolutions.com.au/delphistuf/ftpdemo.htm.

Biography

Roger is a Communications Engineer with 20 years experience in data communication in organisations such as Civil Aviation, Defence Signals and Telstra. His final engagement with Telstra was as Software Architect for the initial Bigpond Cable Internet service. As a principal of Innova Solutions Pty Ltd Roger now concentrates on Delphi developments which utilise his previous experience.