TCP/IP Data Transfer With The NetManage TCP ActiveX

by Bill Fisher

One of the cool add-ons which originally shipped with the Delphi 2.01 maintenance release is the Internet Solutions Pack (ISP). The ISP is a collection of ActiveX controls that Borland licensed from NetManage, a company specializing in Internet connectivity. You might notice that the ISP is fully acronym-compliant, as it includes controls for TCP, FTP, HTML, HTTP, NNTP, POP, SMTP and UDP. These controls can all be found on the Internet page of Delphi's Component Palette, and example code for these controls can be found under the \Delphi 2.0\Demos\Internet sub-directory. In this article, I'll focus on the TCP control.

While the TCP demo application found on the Delphi 2.01 CD is nifty, it only shows how to work with string data. In this article, you'll learn how to build simple TCP projects that send and receive arbitrary binary data. I'll start with the "sender" application entitled SendApp, then move into the "receiving" application called RecApp.

Sender Application

For the purpose of this article, SendApp will broadcast one of three types of data to RecApp: a sound (.wav file), an image (.bmp) and a program (.exe). Of course, it really does not make a difference what the data is, it's just bytes as far as the controls are concerned. To facilitate communication I have a unit shared by SendApp and RecApp that holds handshaking information. It contains the following interface declarations that describe what type of data will be transferred:

```
const
  OneI_BitMap = 0;
  OneI_Wav = 1;
  OneI_EXE = 3;
```

Since the TCP control only knows how to send and receive binary chunks of data, it never knows how much or what type of data it is sending or receiving. Using the record defined below SendApp communicates to the RecApp what kind and how much data they are working with:

```
type
  PTFisherTCP = ^TFisherTCP;
  TFisherTCP = record
    Size: Integer;
    Tag: Integer;
  end;
```

At some point, the RecApp will need to allocate sufficient memory for the data and manipulate the data in a manner dictated by its format. There are a number of ways the sender could communicate this record to the receiver, and I considered the following scenarios: 1) Send the record before each data transfer session, or 2) prepend or append the data with a record, so I know the first (or last) few bytes of data will tell what it is and how big the data is, or 3) send all of the data, and then send an "I am done" signal along with the data at the end of the data transfer (and take some measure to ensure that some pattern of bytes in the data isn't the same as the "I am done" signal). I chose option 1), since it allows me to allocate a fixed-size buffer before the data arrives. It is simple to implement, easy to understand, and it will not cause a major hit on performance.

Regardless of the type of data, for our program all transmissions are routed from one procedure, defined as:

```
procedure SendFisherTCPData(
  Location: Pointer; Size,
  DataType: Integer;
  Tcp: TTcp);
```

➤ *Listing 1*

```
procedure SendTCPData(Location: Pointer; Size: Integer; DataType :
  Integer; Tcp: TTcp);
  procedure FillHeader(PHeader: PTFisherTCP; DataType, Size : Integer);
  begin
    PHeader^.Size := Size;
    PHeader^.Tag := DataType;
  end;
var
  PHeader: PTFisherTCP;
  HeadBuffer, DataBuffer: Variant;
  Ptr: Pointer;
begin
  PHeader := AllocMem(sizeOf(TFisherTCP));
  try
    FillHeader(PHeader, DataType, Size);
    {send the header}
    try
      HeadBuffer := VarArrayCreate([0, SizeOf(TFisherTCP)-1], VarByte);
      Ptr := VarArrayLock(HeadBuffer);
      Move(PHeader^, Ptr^, SizeOf(TFisherTCP));
    finally
      VarArrayUnlock(HeadBuffer);
    end;
    Tcp.SendData(HeadBuffer);
  finally
    FreeMem(Pheader);
  end;
  {Send the Stream}
  DataBuffer := VarArrayCreate([0, Size -1] , VarByte);
  try
    Ptr := VarArrayLock(DataBuffer);
    Move(Location^, Ptr^, Size);
  finally
    VarArrayUnlock(DataBuffer);
  end;
  Tcp.SendData(DataBuffer);
end;
```

The parameters for this procedure are defined as:

- `Location`: a pointer to the start of the data.
- `Size`: the size of the data block pointed to by `Location`.
- `DataType`: the type of data being transferred (so the receiver knows how to deal with it).
- `Tcp`: The TCP control to use to send the data.

I will spend some time examining this procedure because it alone does most the work for the program. All you need to do is get a pointer to your data, pass it along and whoosh! It's sent. The method you use to obtain your data pointer is totally up to you.

`SendTCPData` declares a local procedure which fills in the header record. This is done as a local procedure rather than in the procedure body simply to allow for future expansion of the project, as it really does very little and saves no lines of code. See Listing 1.

Let's take a quick look at the local variables in `SendTCPData`. First, `PHeader` is simply a pointer to the header data record. `HeadBuffer` and `DataBuffer` are data buffers. I use variants, because, being an ActiveX control, the TCP component uses variants for passing data. `Ptr` is used to transfer contents to variants from Object Pascal pointers.

In both of the example programs discussed I'll use a method similar to that shown in Listing 2 for transferring the blocks of memory. On line 2, we allocate memory for the "in" `HeadBuffer` by making a variant array of type `byte`. On line 3, we lock the `HeadBuffer` variant array in memory so it can be copied. On line 4, we copy memory from the pointer to the variant array. On line 6, we unlock the variant array from memory. We wrap the whole thing in a `try-finally` block to make sure the lock is released.

Here is another key point of this example. As most data will be generated with standard Object Pascal types like pointers and classes, the data needs to be converted to variants to work with the TCP control. Also, be careful with the `Move` procedure: it moves blocks of memory with no regard to the areas of memory affected. Because of this, it can cause horrendous memory corruption that can be very difficult to track down, because the corruption may manifest itself in another area of the program or not at all. Pay special attention when using a pointer type as a parameter to move, as that requires use of a caret (^) to de-reference the pointer.

Once the `HeadBuffer` is properly filled, the data can be sent out on the wire with:

```
Tcp.SendData(HeadBuffer);
```

On the other end, `RecApp` will decode the data record and allocate enough memory to store it. It will then quietly sit there and collect the data packets as `SendApp` broadcasts them. The `TTcp` control will handle packet construction, including packet size. The TCP/IP protocol itself will handle all handshaking and packet error correction, including re-sending corrupt packets. So, fortunately, you do not need to be concerned with these issues.

After the data record has been transmitted, it is time to send the actual data. Again, I create a variant array, lock it in memory, copy data into it and unlock the memory. Finally I transmit it exactly as I sent the data record.

As you can see from Listing 1, a few more goodies added like exception handling and proper memory management are also added.

As I explained earlier, the method used to obtain a pointer to the data and the data's size is inconsequential. The methods I used, however, may serve to quickly demonstrate a couple of ways to get at your information. I started my testing with a bitmap and I suggest that you do as well. Bitmaps are forgiving when it comes to corruption: problems will often manifest themselves as visible glitches in the bitmap rather than annoying system crashes. Also, bitmaps themselves contain information about the image loaded at the beginning, but no footer data. This works well if your data gets truncated or corrupted after the first packet. In my case, I use a `TImage` to hold the image I want to transfer. In this example, I'll move the image into a memory stream in order to help demonstrate how to send a `TMemoryStream`. All in all, the process for calling `SendFisherTCPData` is rather mundane. See Listing 3.

After sending some bitmaps successfully, it's time to move onto a slightly more complex type of data. Wav files make good test data too. This format lends itself to detecting whether footer or other trailing information is somehow truncated. Again, I employ a `TMemoryStream` to send the data. See Listing 4.

Lastly, I will transfer an actual program (Listing 5). Once the initial transfer types are working

Listing 2

```
try
  HeadBuffer := VarArrayCreate([0, SizeOf(TFisherTCP)-1], VarByte);
  Ptr := VarArrayLock(HeadBuffer);
  Move(PHeader^, Ptr^, SizeOf(TFisherTCP));
finally
  VarArrayUnlock(HeadBuffer);
end;
```

Listing 3

```
procedure TSendMain.SendBitMap1Click(Sender: TObject);
var
  Stream : TMemoryStream;
begin
  Stream := TmemoryStream.Create;
  try
    SendMain.Image1.Picture.Bitmap.SaveToStream(Stream);
    Stream.Seek(0,0);
    SendTCPData(Stream.memory, Stream.Size, OneI_BitMap, Tcp1);
  finally
    Stream.Free;
  end;
end;
```

satisfactorily, it is very little effort to add more data types to the transfer mechanism.

Before going onto the "receiver" application, there is one more feature I would like to add. The TTcp control does not have an OnStateChange event, which would be necessary to determine when to update the connection's state within the user interface.

One way to solve this problem is to have a TTimer fire off periodically to ask the TTcp control about its current state. The OnTimer event will make a call to the GetTcpState function (Listing 6) which will return a string value that can be used in the user interface. Fortunately, the control does have a state property of type integer, so I use a simple case statement to return the desired string.

While discussing the State property, I'd also like to point out another problem that cropped up during development of this application. Occasionally, when terminated, it would generate an Access Violation.

It seems that the TCP control doesn't completely de-initialize itself even after its Close method is called and its State property is checked to insure the control is actually closed. To get around this, just add a little loop to pause after closing. Admittedly, this is not the most elegant solution in the world, although it does do the trick. This problem did not surface in the SendApp application, but it did in RecApp where I had controls in a "listening" state.

Receiving Application

As it turns out, RecApp has a few more pitfalls than SendApp. I found a problem immediately when trying to design the application using just one TCP control. This won't work, as the TCP controls cannot handle connection requests all by themselves.

So we need one TTcp control whose job is to simply listen for connection requests and then hands those requests to a second TTcp control which services the connection. This takes place in the listening control's Connection Request event handler. The "listener" will invoke the "handler" control's Accept method and the "listener" will then close.

This dual control design is necessary because only one TCP control at a time can handle connections. So until that "handler" is finished, the application should refuse all other connection attempts. Closing the "listener" achieves this, without forcing the new connection attempt to wait while the application tries to figure out which control should handle an incoming connection. See Listing 7.

Once a connection has been accepted by a TCP control, its DataArrival event is fired off repeatedly until the data transmission is complete. At that point both controls should be reset and wait for another connection request.

➤ *Listing 4*

```
procedure TSendMain.Sendwav1Click(Sender: TObject);
var
  DefaultFile : String;
  Stream : TMemoryStream;
begin
  Stream := TmemoryStream.Create;
  try
    DefaultFile := ExtractFileDir(ParamStr(0)) + '\' + 'ah.wav';
    Stream.LoadFromFile(DefaultFile);
    Stream.Seek(0,0);
    SendTCPData(Stream.Memory, Stream.Size, OneI_Wav, Tcp1);
  finally
    Stream.Free;
  end;
end;
```

➤ *Listing 5*

```
procedure TSendMain.SendEXE1Click(Sender: TObject);
var
  DefaultFile : String;
  Stream : TMemoryStream;
begin
  Stream := TmemoryStream.Create;
  try
    DefaultFile := ExtractFileDir(paramstr(0)) + '\' + 'OneEye.exe';
    Stream.loadFromFile(DefaultFile);
    Stream.Seek(0,0);
    SendTCPData(Stream.memory, Stream.size, 3, Tcp1);
  finally
    Stream.Free;
  end;
end;
```

➤ *Listing 6*

```
function GetTcpState(Tcp : TTcp): string;
begin
  case Tcp.State of
    0:  Result := 'Closed';
    1:  Result := 'Open';
    2:  Result := 'Listening';
    3:  Result := 'Connection is Pending';
    4:  Result := 'Resolving the host name';
    5:  Result := 'Host is Resolved';
    6:  Result := 'Connecting';
    7:  Result := 'Connected to ' + Tcp.RemoteHost;
    8:  Result := 'Connection is closing';
    9:  Result := 'State error has occurred';
    10: Result := 'Connection state is undetermined';
  end;
end;
```

➤ *Listing 7*

```
procedure TMainForm.TCP2ConnectionRequest(Sender: TObject; requestID: Integer);
begin
  {Note: these TCP controls can not issue an accept() to
   themselves. So there must be two controls}
  Tcp1.Accept(requestId);
  Tcp2.Close;
  caption := 'Connected';
end;
```

The DataArrival event will be called each time packets are received. The code is in Listing 8.

Because of the nature of TCP/IP, we don't know how many times this event will fire, regardless of the data size. Even if it was fired six times during the transmission of some particular set of data, there's no guarantee it will be called six times the next time you transmit that data. The application must handle the transaction in such a way that you can save the previous data and append the current data. All you can count on is that the DataArrival event will fire one or more times during transmission.

Before getting deeply involved, let's lay out the game plan. In the DataArrival event handler, I'll use global variables to determine if we are receiving actual data (not a record header). Using a header, I'll know the size of the data transmission before I receive the data. In this way, I can allocate enough memory to store all of the data, not just this packet, beforehand.

I employ these global variables inside DataArrival in order to remember that state of the data transfer from a previous call to DataArrival. With past versions of Borland Pascal, I may have used typed constants instead of global variables. With Delphi 2, Borland is moving away from assignable typed constants (constants that can change), even adding a compiler switch {J+} to prevent doing this.

The global variables are utilized in my implementation section and are defined as:

```
var
  count : integer = 0;
  pheader : PTFisherTCP = nil;
  head : pbyte = nil;
```

I can also use these variables to determine if the control is starting a new transaction, or getting more data from an existing transaction. If it is a new transaction, then I know this is a record header and can confidently extract data to determine the size of the data transfer, allocate memory for the data and set the constants to receive data. Otherwise I just append this packet to the data buffer.

A key point of DataArrival is it does not actually get data, but it instead informs you that data is now available and offers the only chance to copy the data from the current packet. If you do not get the data now the next packet will overwrite this packet and the data will be lost. To grab this crucial data, you call the control's GetData method, declared as:

```
procedure GetData(
  var Data: Variant;
  const Type_,
  MaxLen: Variant); stdcall;
```

Data is the variant buffer that will hold the current packet's information. Type_ describes what type of variant Data is (ie the type of information placed in Data (see the Help for the different types the NetManage control can recognize). MaxLen is the size of the data waiting to be picked up. Another pitfall: once you call GetData, the BytesTotal variable for DataArrival will be cleared! So if you're adding up the bytes received, do it before you call GetData.

Back to the game plan. The variable Head is used as a pointer to the beginning of the data buffer allocated in memory. Another pointer will be used to track the current position within the data buffer. Think of this second pointer as a window into your full data buffer.

Each time new data arrives, I write to the current position "window" and then move the "window" pointer forward by the number of bytes just written to memory.

Because variants are used to get data from the TTcp control, it will need to be converted to Object Pascal types in order for Delphi to manipulate the data natively.

➤ *Listing 8*

```
procedure TMainForm.TCP1DataArrival(Sender: TObject; bytesTotal: Integer);
var
  Window : pbyte;
  Ptr: pointer;
  DataBuffer, HeaderBuffer: variant;
begin
  if pheader = nil then begin   // new data !!
    {New header, get setup}
    pheader := AllocMem(SizeOf(TFisherTCP));
    HeaderBuffer := VarArrayCreate([0,SizeOf(TFisherTCP) -1], varbyte);
    {grab the header record}
    Tcp1.GetData(headerbuffer, (VT_Array or VT_ui1) , SizeOf(TFisherTCP) );
    Try
      {copy the data to my header variable}
      ptr := VarArrayLock(headerbuffer);
      move(Ptr^, pheader^, SizeOf(TFisherTCP));
    Finally
      VarArrayUnlock(HeaderBuffer);
    end;
    {finally allocate a buffer for the data, using a head pointer}
    head := AllocMem(pheader^.size + 1 * sizeof(byte));
    caption := 'Receiving';
  end else begin
    { pheader is not nil, means we are grabbing data! }
    { set up my data structures}
    DataBuffer := VarArrayCreate([0,BytesTotal], varbyte);
    {set my pointer}
    window := head;
    inc(window, count);
    {grab a chunk of data from the port}
    inc(count, Tcp1.BytesReceived);
    Tcp1.GetData(DataBuffer, (VT_Array or VT_ui1 ) , BytesTotal);
    {copy that hunk of data over}
    try
      ptr := VarArrayLock(DataBuffer);
      move(Ptr^, window^, BytesTotal);
    finally
      VarArrayUnlock(DataBuffer);
    end;
    {At the end of each data transfer, check to see if this was the last
     transfer...if so, use the data somehow}
    if count  = pheader^.size then begin
      UseData(head, count, pheader^.tag);
      {Reset the data connection for the next transfer}
      FreeMem(pheader);
      pheader := nil;
      count := 0;
      FreeMem(head);
      caption := 'Transfer Complete';
    end //if-count
  end;  //if-else..(pheader = nil)
end;
```

Fortunately the process is almost identical to the system used in SendApp. I'll need HeadBuffer and DataBuffer to be variants. Sharing one is an option, but I will use two variants and keep this as simple as possible. These two variables will correspond to SendApp's HeadBuffer and DataBuffer variables. Again a pointer is needed to lock down the variant memory that comes in and store it to the Head variable (see the var section in Listing 8).

As mentioned before, the DataArrival event in RecApp will be called multiple times. In any given iteration, I can tell if there is a new transaction based on the value of PHeader. PHeader is initialized to nil, which indicates there is a new data transmission and a data header is being received.

I will allocate memory for my header and set up the variant array to collect the header file from the TTcp.GetData method. I will then lock down the memory and copy, much as I did in SendApp.

Once the header record is transferred and copied from a variant to a more "conventional" memory buffer, I can allocate enough memory for the entire transaction.

If PHeader was not nil, then I know I have already received the header and am getting data for the first time, or getting more data from an existing transaction: it does not matter which. First, I allocate memory for the variant that GetData will use. Then I move the write "window" along by the current count. After incrementing the location of the "window" I add the current number of bytes to count.

Next I request a binary packet from the TTcp control's port. Another pitfall: what is the correct data type to use for the variant? The control wants an array of bytes, but there is no variant type for a byte array *[If you are sitting there chewing your fingernails because you never worked with variants before, don't fret, just check out Xavier Pacheco's excellent article on the subject in Issue 13, September 1996. Editor]*. The solution is to declare the type as a combination of VT_Array and VT_UI1 like this:

```
Tcp1.GetData(
  DataBuffer,
  (VT_Array or VT_ UI1 ),
  BytesTotal);
```

Finally, I lock down the memory and copy it over as shown earlier.

For each transmission, I need to check if the current batch of data completes the current transaction. To do this, I check to see if the header record Size field is equal to the Count variable. If it is, this transaction is complete. I'll then reset the variables and constants to handle a new transaction. Of course I'll also need to put the data I just received to work in some way. In this simple case, I'll keep it to a single procedure named UseData.

If this was not the last bit of data in the current transmission, then the DataArrival event will fire again. All the variables have been left in the state to receive more data and append it to the data already received, so there will be nothing more to do.

Assuming all goes according to plan, the data is now in the receiving application. Using a modular approach, I pass the received data on to a procedure called UseData for processing (Listing 9). It takes a pointer, the size of the data pointed at and the type of information being pointed to. After that I use a case statement in the body of the procedure to perform different processing based on the value of DataType.

In the first case, I'll load a bitmap into a TImage and resize the main form to match the size of the TImage. This is fun, but kind of dangerous, as bitmaps can be smaller than the legal size of a form. So always keep in mind your incoming data may not be 100% compatible with your data handlers. For a .WAV sound, I simply play it from memory using the PlaySound API function. Finally, for the executable I use a TFileStream to write it to disk and fire off CreateProcess to launch the program.

➤ *Listing 9*

```
procedure UseData(Head : PByte; Count, DataType: Integer);
var
  Stream: TMemoryStream;
  FileStream: TFileStream;
  StartUpInfo: TStartUpInfo;
  ProcessInfo: TProcessInformation;
begin
  case DataType of
    OneI_BitMap :
      begin
        Stream := TMemoryStream.Create;
        try
          Stream.Write(Head^, Count);
          Stream.Seek(0,0);
          with MainForm do begin
            Image1.Picture.Bitmap.LoadFromStream(Stream);
            Width := Image1.Picture.Bitmap.Width;
            Height := Image1.Picture.Bitmap.Height;
          end;
        finally
          Stream.Free;
        end;
      end;
    OneI_Wav :
      begin
        if not PlaySound (pChar(head), 0, snd_memory) then
          ShowMessage('Received a .wav that could not played')
      end;
    OneI_Exe:
      begin
        FileStream := TFileStream.Create('c:\sample.exe', fmCreate);
        try
          FileStream.write(head^, Count);
        finally
          FileStream.free;
        end;
        FillChar(StartUpInfo, SizeOf(TStartUpInfo), 0);
        with StartUpInfo do begin
          cb := SizeOf(TStartUpInfo);
          wShowWindow := SW_ShowNormal;
        end;
        CreateProcess('c:\Sample.exe', Nil, Nil, Nil, False,
          NORMAL_PRIORITY_CLASS, Nil, Nil, StartupInfo, ProcessInfo);
        {might want to delete the file now - might not...}
      end;
  else
    ShowMessage('What is this Data?');
  end; //case
end;
```

Conclusion

In the past you might have thought that sending custom data over a TCP/IP network was a complicated task reserved for gurus, but as you can now see the Internet Solutions Pack TCP control brings custom TCP/IP programming to everyone. This article demonstrates how to transmit three different types of custom data over the wire, but it should be clear how to apply the techniques I've demonstrated to any type of data. Heck, you could even use these techniques to share data with friends over the internet. Keep the amount of data small, though, it's not nice to suck up all that bandwidth better used for playing Decent...

Bill Fisher is a Senior Engineer with Borland's Delphi Developer Support department. He has been with this group since the early days of Delphi 1.0. In addition to his primary customer support role, he is involved in recruiting, interviewing and training new Delphi team members. You can contact him at: wfisher@corp.borland.com

Bill would also like to thank Steve Teixeira for his editing of and and input to this article. Steve is a Software Engineer in Borland's Delphi R&D department and the co-author of *Delphi 2 Developer's Guide* from SAMS publishing. You'll find his articles published in *The Delphi Magazine*. Say howdy to Steve at steixeira@corp.borland.com.