

Internet Direct (Indy) - Servers

by Hadi Hariri

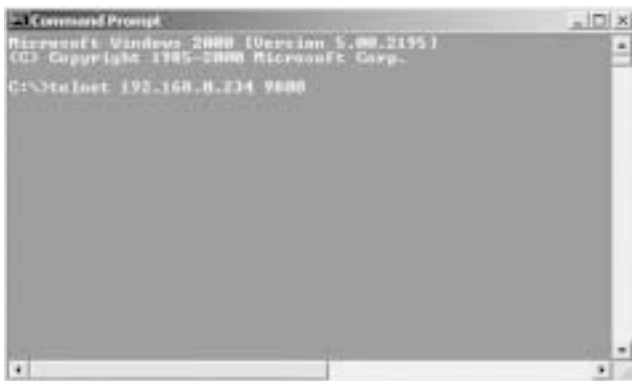


In the previous issue, Allen O'Neill gave us a brief introduction to Internet Direct, and showed us some concepts of both TCP/IP programming and Indy. In this article we cover the basics of servers and how they are implemented, the Indy way!

Chicken or Egg?

One of the problems faced when writing about TCP/IP programming is whether to start by explaining Servers or Clients. It is the classic paradox of which comes first, the chicken or the egg? On the whole, clients can be considered easier to understand and to implement than most servers. However, one of the drawbacks of implementing clients is that you need a server to test against. This does not cause headaches when you are developing a known protocol such as HTTP or FTP, but when working with custom protocols it is very hard to know if you are on the right track and most of the time you are programming with a blindfold on.

We could go about explaining clients first by developing our own mail application (SMTP/POP protocols) but we would then have to skip the basics of TCP and jump directly to the higher abstract classes that implement protocols, and during the course of this, miss out on some of the most important concepts related to TCP/IP. That is why we will show the basics by implementing servers.



Most protocols are text based, that is, commands are interchanged between the client and the server using plain English. For example, the HTTP client protocol uses commands such as *GET*, *PUT*, *POST* to indicate to the server what it requires. This provides us with an enormous advantage; by developing a text-based protocol we can use TELNET to act as our client and run tests. TELNET is provided on most, if not all platforms. Under Win32 and Linux platforms you can start a telnet session by just typing *telnet* from the command prompt, followed by the IP address of the server you wish to connect to. In itself, TELNET is a protocol and by default it connects to port 23. We can optionally override the default by specifying a port following the IP address.

TCP/IP

The term TCP/IP (Transfer Control Protocol / Internet Protocol) is somewhat confusing to the newcomer. It is used to denote the network as a whole. Services or

applications on a TCP/IP network can use either the TCP or UDP protocol. The main difference between the two is that TCP is a connection-oriented protocol whereas UDP is connectionless. Comparing the two to a real-world analogy, we can come up with the telephone and pager example. When one uses a telephone, one has to first dial a number (establish a connection) and then talk. With a pager, a message is simply sent to a recipient without the need of establishing a prior connection, and there is generally no guarantee that the message has been received. There are other differences between the two shown in the table below:

| TCP | UDP |
|-------------------------------|---------------------------|
| Connection-oriented | Connectionless |
| Reliability of packet arrival | No guarantee |
| No duplicates | Duplicates may be sent |
| A lot of overhead | Fast, very small overhead |

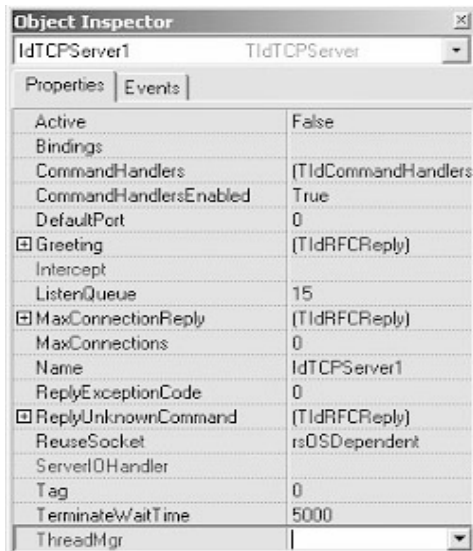
If UDP is unreliable and a transfer can contain duplicate packets, why use it? There are a lot of applications that are based on the UDP protocol, such as RealPlayer, ICQ, etc. The reason for this is that it is much faster than TCP. Applications that do not require 100% reliability can use UDP. One would not even notice if a couple of frames were lost during an audio stream or if an ICQ message was misplaced or sent twice. Since most applications, however, are based on TCP, we are going to examine this protocol first.

Indy provides two options for building TCP servers: *TidTCP*Server and *TidSimple*Server. In real-life, the latter is hardly ever used; reason being that it can only handle one client connection at a time. There is not much point in building mail servers or web servers that only allow one simultaneous connection. However, in some circumstance, where you know that you will only ever have one connection at a time, the *TidSimple*Server is more than sufficient.

TidTCP

When you first drop a *TidTCP*Server on the form, you will be overwhelmed with the amount of properties that it contains. Fear not though, most of these properties only come into play when using what are known as *CommandHandlers*. *CommandHandlers* make building TCP servers child's play. In this first approach we will use the "traditional" way of building a server and later on see the equivalent by using *CommandHandlers*.

As seen in the previous article, each server is identified by an IP and port number. This allows for different servers to run on the same machine and offer different services depending on the port the client connects to. The *DefaultPort* property identifies the port on which the server will run. Ports under 1024 are reserved by the operating system and should not be used unless you are building a server for a known protocol (for example HTTP). In fact, in Linux, if you specify a port below the permitted values, you have to run the application as root. For custom servers, you should use ports 1025 and over.



Apart from assigning a port, you can also assign an IP address on which the server will be available. If a machine has more than one IP address, for example, 192.168.0.1 and 192.168.0.2, you can tell the server to only run when the client connection is established to 192.168.0.2. This can be accomplished by entering the IP/Port combination in the *Bindings* property. It is very important to make sure you do not have invalid information in this property before you deploy or test your application on a different machine.

If you set the IP address to your development machine, the chances are that your application will not run on the target machine unless it has the same IP address. The *Binding* property also allows you to specify additional ports on which the server can listen. Apart from being used for straight protocols, bindings can allow you to build a “bridge” between two subnets, where connections on one IP are routed out/back via the other IP. The other relevant property right now is the *Active* property. This is the “switch” for the server, by setting it to *True*, the server will start listening for incoming connections.

The three most important events used when implementing servers based on *TIdTCPServer* are:

- OnConnect
- OnDisconnect
- OnExecute

The first event is fired when a new connection is made to the server, and the second when a client disconnects. Each server needs to implement a service, otherwise it would be quite useless. With Indy, this “service” is coded in the *OnExecute* event of the *TIdTCPServer*. This event is fired each time a connection is established to the server.

```
procedure
    TForm1.IdTCPServer1Execute(AThread:
        TIdPeerThread);
begin
    // Implement service here
end;
```

Imagine a server that responds to a client connection with the current date:

```
FormatDateTime('dd mmm yyyy', Date);
```

When the first client connects, the server responds with the current date. When the second client connects, it would have to wait for the first client to complete. This does not cause a problem if the service is simple (like the previous example), but processes that take various seconds or minutes to complete would cause bottlenecks for incoming clients; needless to say, the server would be very inefficient. One solution to this problem is to have each client connection run in its own thread independently of other clients. Indy takes care of this by passing a thread parameter (*AThread*) to the *OnExecute* event. **The *OnExecute* is executed in the context of the *AThread* parameter.** This is very important to remember since the code placed in the *OnExecute* has to be **thread-safe**. The *AThread* can be thought of as the client; it does not represent only the connection, but the client as a whole. All communication with the client, and information about the client itself (*such as IP address, etc*) are all accomplished through the ***AThread* parameter**.

Bugs'r'us

Our first server application is going to consist of a bug-tracking tool. We will first implement the server using the “normal” or classic Indy approach of using the *OnExecute* method and in a later article we will see how to do the same using Command Handlers.

The server has no GUI interface; the main form consists solely of a *TLabel*. That is all we need since we will not be interacting with the server as such. Most of the code, if not all, is located in the *OnExecute* which we’ll examine in detail shortly however, before implementing any service or custom protocol, we need to define it. In this case, our protocol is text based and consists of the following commands: ADD, DELETE, MODIFY, STATUS, DISPLAY, LIST, HELP and QUIT. Some of these commands are followed by a parameter, which indicates a bug identification number.

Each bug is stored as a text file in the *data* directory under the main program folder. Bug entries are represented as simple *StringLists* and are saved directly to the file with the same format. This format consists of the fieldname followed by an equal sign and the value:

```
Title=VB IDE
Description=Major bug in Visual Basic.
                Cannot program! ... Go
                Home !!

Status=Open
```

In addition to the text files, we also use an INI file to store the server settings and the list of bugs. To help with reading and writing to this INI file we use an auxiliary class named *TBugSettings*.

```

TBugSettings = class(TObject)
protected
    FIniFile: TIniFile;

    function GetLastID: Integer;
    function GetPort: Integer;
    function GetVersion: string;
public
    constructor Create;
    destructor Destroy; override;

    procedure AddBugEntry(ABugID:
        string);
    procedure DeleteBugEntry(ABugID:
        string);
    procedure BugList(AList:
        TStringList);

    property LastID: Integer read
        GetLastID;
    property Port: Integer read
        GetPort;
    property Version: string read
        GetVersion;
end;

```

The implementation is no mystery. It simply consists of methods to read (and in some cases write) to the INI file. The advantage is that we can then access the values directly in our server using simple properties. *AddBugEntry*, *DeleteBugEntry* and *BugList* are helper methods to maintain a list of the current bugs and prevent us from having to enumerate the files in the *data* directory. *LastID* returns the last ID (auto-incremental number) used for a bug entry. It returns the value and increments it by 1, writing the new value to the INI file, ready for the next access. *Port* indicates the port on which our server will run (by default 10000).

The start-up and shutdown of the server takes place in the *OnCreate* and *OnClose* of the main form respectively.

```

procedure TMainForm.FormCreate(Sender:
    TObject);
begin
    GDataPath :=
        ExtractFilePath(ParamStr(0))
        + 'DATA\';
    if not DirectoryExists(GDataPath)
        then begin
            ForceDirectories(GDataPath);
        end;
    BugServer.DefaultPort :=
        GSettings.Port;
    BugServer.Active := True;
end;

procedure TMainForm.FormClose(Sender:
    TObject; var Action:
    TCloseAction);
begin
    BugServer.Active := False;
end;

```

In the *OnCreate* we check to see if the *data* directory exists and if not, we create it. We then set the port for the server and activate this. Before closing the main form, we deactivate the server by setting its *Active* property to false.

GSettings, which represents our global variable for accessing server settings, is created and destroyed in the *initialization* and *finalization* code respectively.

Before we go on to examining the main skeleton of our server (the *OnExecute* event), there is one piece of code left to do: the *OnConnect*. Here we simply respond to the client with a text message welcoming the connection and providing the server version.

```

procedure
    TMainForm.BugServerConnect(AThread:
        TIdPeerThread);
begin
    AThread.Connection.WriteLine('Welcome
        to BugServer Version ' +
        GSettings.Version);
end;

```

AThread which is of type *TIdPeerThread* is passed as a parameter in the event handler. As mentioned before, it represents the client. The most important property of this class is the *Connection*. This class has various methods to read and write from the connection. The most common one is *WriteLn* which takes a string parameter and writes it to the connection, appending an EOL.

The Server Process

When a client connects to the server, and after the *OnConnect* is fired, the server then proceeds to the *OnExecute* event. All communication with the client takes place here while the client is connected. The first thing we need to do is read what command the client has asked us for. To do this, we again use the *AThread.Connection* class. In this case, we call the *ReadLn* method. Like its counterpart, *ReadLn* blocks on the call and waits for the client to send a line of text. It is important to remember that the call **blocks**, that is, the server pauses its execution on this instruction waiting for text from the client. Since we have two types of commands, one with parameters and one without, we first need to divide the string sent to us from the client into the command and parameter (if one exists):

```

LCommand := AThread.Connection.ReadLn;
if Pos(' ', LCommand) > 0 then begin
    LParams := LCommand;
    LCommand := Fetch(LParams, ' ');
end;

```

Once we have our command and optionally the parameter, by using a simple *if* structure, we examine what the command is and take the appropriate action.

```

if AnsiSameText(LCommand, 'QUIT')
    then begin
        AThread.Connection.WriteLine('Goodbye');
        AThread.Connection.Disconnect;
    end else if AnsiSameText(LCommand,
        'HELP') then begin
        with AThread.Connection do begin
            WriteLn('200 HELP Follows:');
            WriteLn;
            WriteLn('ADD           - Add a
                new bug');
            WriteLn('DELETE bug_no - Delete
                bug_no');
        end;
    end;

```

```

WriteLn('MODIFY bug_no - Modify
      bug_no');
WriteLn('STATUS bug_no - Get
      the status of bug_no');
WriteLn('DISPLAY bug_no -
      Display bug information
      of bug_no');
WriteLn('LIST          - List
      all bugs');
WriteLn('QUIT          -
      Quit');
WriteLn('HELP          - This
      screen');
WriteLn('.');
end;

```

If the client sent the *QUIT* command, we send back the text *GoodBye* and disconnect the client. At this point, the *OnExecute* event will **never** be called for that client again until it reconnects. This is how to finish communication with the client from the server side, by calling the *AThread.Connection.Disconnect* method. If the *HELP* command was requested, we simply respond with various lines of text indicating what each command does. We finish this by sending a "." on its own. The "." acts as a delimiter. This way, when implementing the client, we know that after sending a *HELP* command, we have to read from the connection until we encounter a "." on a line on its own.

```

end else if AnsiSameText(LCommand,
  'ADD') then begin
  LBug := TStringList.Create;
  try
    with AThread.Connection do begin
      WriteLn('200 ADD End input
        with . on blank line');
      Capture(LBug);
      LBugID := GSettings.LastID;
      LBug.SaveToFile(GDataPath +
        IntToStr(LBugID) +
        '.txt');
      GSettings.AddBugEntry(IntToStr(LBugID));
      WriteLn('200 OK');
    end;
  finally
    FreeAndNil(LBug);
  end;
end else if AnsiSameText(LCommand,
  'DELETE') then begin
  if FileExists(GDataPath + LParams
    + '.txt') then begin
    DeleteFile(GDataPath + LParams +
      '.txt');
    GSettings.DeleteBugEntry(LParams);
    AThread.Connection.WriteLn('200
      OK');
  end else begin
    AThread.Connection.WriteLn('500
      Invalid bug number');
  end;
end;

```

The next two commands are *ADD* and *DELETE*. In the case of *ADD*, the client has to send the information about the bug in the format *FIELDNAME=FIELDVALUE*. Each field is represented on its own line. To end the transmission, the client sends a "." on its own (similar to what was done on the server side with the *HELP* command). This protocol of

ending information with a "." is very common and widely used in many Internet RFCs (for example: *SMTP*). For this reason, Indy already has a method to handle this; the *Capture* method reads information from the connection until it encounters a ".". It takes a *StringList* as its parameter, and pushes all the information it receives into that *StringList*. In the code above, we send the client an acknowledgement saying that it can start to send the information. We then capture this information into the *StringList* and respond to the client with a *200 OK* message if everything went well. Again, this last message to the client will later help us to develop the client since we will know that the protocol will respond with *200 OK* if there were no errors.

The *DELETE* command is our first command that requires an additional parameter. This parameter indicates the bug number (ID) that we wish to delete. We should already have the value in the *LParam* variable. We check to see if this bug exists and delete it if it does. We then delete the entry from our INI file by using the *DeleteBugEntry* method. Again here, if the process was successful, we reply with a *200 OK*. On the other hand, if the bug number does not exist, we reply with a *500 Invalid bug number* message.

The *MODIFY* command can be thought of as a combination of the previous two. It first checks to see if the bug we want to modify exists and, if so, asks for the information. It then captures this information and replaces the values that have changed. *STATUS* returns the status of the bug passed in as a parameter by simply loading the bug into a *StringList* and returning the *STATUS* field value.

DISPLAY uses a new method, *WriteStrings*. This procedure, writes a series of strings to the connection. Optionally it can write the number of strings it is going to send when the protocol requires it. In our case we do not require this because we are ending our transmission with a ".". However, if our protocol were to first send the number of lines, on the receiving end we would first need to read this number and then issue the corresponding number of *ReadLn*'s.

```

end else if AnsiSameText(LCommand,
  'DISPLAY') then begin
  with AThread.Connection do begin
    if FileExists(GDataPath +
      LParams + '.txt') then
      begin
        LFile := TStringList.Create;
        try
          LFile.LoadFromFile(GDataPath
            + LParams + '.txt');
          WriteStrings(LFile);
          WriteLn('.');
        finally
          FreeAndNil(LFile);
        end;
      end else begin
        WriteLn('500 Invalid bug
          number');
      end;
    end;
  end;
end;

```

LIST is very similar to the previous, except that instead of reading from the text file it reads from the *GSettings* object that contains a list of the bug entries.

```

end else if AnsiSameText(LCommand,
    'LIST') then begin
    LBug := TStringList.Create;
    try
        GSettings.BugList(LBug);
        AThread.Connection.WriteString(LBug);
        AThread.Connection.WriteLine('.');
    finally
        FreeAndNil(LBug);
    end;
end else begin
    AThread.Connection.WriteLine('500
        Invalid Command');
end;
end;

```

If the command sent does not coincide with any of the previous, we simply reply with a *500 Invalid Command*.

What you can notice from the server code is that it is very long and messy. Obviously we could have split the code up into separate procedures and called each one, depending on the command sent to the server. However, the reason I left it like this is so that we can later compare the code with that of using Command Handlers. We will see how this reduces the amount of code significantly and that a lot of the manual work is accomplished automatically.

The most important thing to remember when designing your own protocol is that both client and server have to be implemented. Therefore, it is important to specify the necessary steps that indicate when a transmission starts, when it ends and how it ends. This is the reason that we used reply codes such as *200 OK* or *500 Invalid*. This will allow the client to know how the transmission ended. Many existing protocols such as HTTP or FTP use this method to communicate. We will see that, when using Command Handlers, most of this is taken care of for us also.

Testing the server

Before building the server, make sure that you have the latest release of Indy (version 9). There are a number of properties and methods that have been added to the base classes which did not exist in version 8. Version 9 is still in beta phase but it is quite stable and in fact is being used in numerous production environments.

Although we do not have a client yet, we can test the server using *TELNET*. To do so, open up a telnet session and connect to the local machine by typing *telnet localhost 10000*.

```

Command Prompt - telnet localhost 10000
Welcome to BugServer Version 1.00
HELP
200 HELP Follows:
ADD          - Add a new bug
DELETE bug_no - Delete bug_no
MODIFY bug_no - Modify bug_no
STATUS bug_no - Get the status of bug_no
DISPLAY bug_no - Display bug information of bug_no
LIST         - List all bugs
QUIT        - Quit
HELP        - This screen

```

If the server is running, you will be presented with the welcome message. To see how the server works, simply type in a command that has been implemented.

Each command will display a different output depending on its nature. For example, typing the *HELP* command will produce a multi-line output with information on the commands available on the server.

In the next issue, *Allen O'Neill* will talk about clients and will provide a simple client implementation for the server protocol we have just seen. He will also cover an important issue in Indy, IOHandlers. Make sure you do not miss the next article.

Hadi Hariri is a Software Engineer at Nevrona Designs. He is also Project Co-coordinator for Internet Direct (Indy), the Open-source project of TCP/IP components that is included in both Kylix and Delphi 6. Having formerly worked for an ISP and software development company, he has extensive knowledge with Internet and Client/Server applications as well as network administration and security. Hadi is married and lives in Spain, where he has been a major contributing author to a leading Delphi magazine. He is also co-author of the upcoming book "Delphi Developer's Guide to Internet Direct" and has spoken at Borland Conferences and user groups.