Internet Direct (Indy) an Introduction

by Allen O'Neill

Internet Direct (commonly known as Indy) has emerged over the past year as the defacto TCP-IP/Sockets programming library for Borland compilers. Freely available and developed under OpenSource licensing, this set of classes is now included in Delphi 6 and Kylix distributions. In this introductory article, Allen O'Neill, who is actively involved with Indy, gives us a brief glimpse into the vast ocean of useful classes included with Indy, and gives us a tantalising taste of the power now available to use for TCP and UDP programming in Delphi, Kylix and C++Builder.

Background

Those who have copies of either Delphi 6 or Kylix cannot failed to have noticed the three new tabs in the component palette, literally bulging with cute enticing icons. There are currently over 80 components and over 100 useful classes that make up Indy, covering every major Internet based protocol from plain vanilla TCP, through SNTP, DNS, POP3, Rsh, and IRC, to name but a few. Indy allows you to develop TCP and UDP communications applications that work just as well on your local network as they do across the Internet. As an aside, I should point out that the default version that is distributed with Delphi 6 and Kylix is version 8.0. Indy has since moved onwards and upwards to version 9.0 – you are well advised to upgrade if you have not done so already.

Indy in its infancy was known as *winshoes* - being a pun on winsock - and, as winshoes, it commanded a strong and dedicated following among developers worldwide. When the winshoes core development team learned that Borland were planning Delphi for Linux, they decided to redevelop from the ground up, and create a set of socket classes that would be truly cross platform. Through careful planning and intelligent guesswork, the team was able to deliver a set of classes that compiled successfully the very first time they were tried with Kylix.

One of the arguments I come across in both commercial and in-house development, is whether one should stick to vanilla classes as provided by Borland, or use third party alternatives. Thankfully, with Borland's inclusion of Indy directly on the default palette, this decision has been made for us; we get the best of the OpenSource development model, and Borland's default distribution of the suite ensures that we are not stepping outside the box. Oh, and yes, it was decided that the *win* part should be dropped both in deference to the Linux community, and to reflect its now true cross platform operability!

The provision of this set of communications classes that operate without any change whatsoever on both the Windows and Linux platform, gives us a very powerful tool indeed in our development arsenal. As will be outlined in this and upcoming articles, there is almost nothing one cannot achieve with TCP using Indy and a little imagination.

Alternatives

Since Delphi 2, the default TCP offering by Borland was the Netmasters suite. Rather than get political, suffice to say that the appearance of Indy as part of the default distribution in Delphi and Kylix has brought a sigh of relief to many development houses all over the world. Indy, of course, is not the only TCP suite available; there are fine offerings out there by Francois Piette, and Stefan Hoffmeister to name but two. Indeed, Francois' set of classes called *MidWare* is the architectural backbone for dbOvernet, an nTier alternative provided by Dalco Technologies in the US. There are many differences between Indy and the other offerings available (both OpenSource and commercial), but the most glaringly obvious is the comprehensive inclusion of so many different protocols in Indy.

Diagram (A) illustrates some of the available classes and protocols:

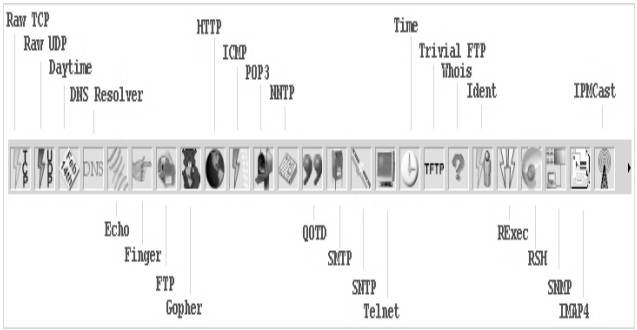


diagram (A) – available client classes and protocols

As you will notice from the small arrow to the right of the diagram (A), there are actually more components than will fit on the screen comfortably, and this is only the clients palette.

The missing clients icons are:

SNPP, IRC, LPR, DayTimeUDP, EchoUDP, QOTDUDP, SysLog and TimeUDP.

(There is the same number of components again on the servers and miscellaneous palette!)

For each client, there is a corresponding server. This means that not only can you, for example, construct your own FTP client, but also your own FTP server. This is extremely handy when you want to implement a custom protocol that adds little to an existing Internet standard; in the FTP example, perhaps you may wish to add version control to files being uploaded and downloaded.

Apart from the multitude of available classes, the main difference between Indy and its counterparts is the architecture. Indy is designed using a blocking architecture that lends itself extremely well to both threaded and non-threaded systems. I will cover the merits and politics of blocking versus non-blocking in a future article, suffice at the moment to issue the immortal words of Chad Z. Hower *(*the originator of Indy*)* – 'Blocking is not evil!!'

Blocking essentially means that, once you issue a command, you must wait for it to return a value or do something before it returns control to you. This is distinct from non-blocking where you can issue a command, do something else, and monitor an incoming system message to retake control of the process. Blocking works in a similar fashion to working with file. This is clearly illustrated in code listing (A).

Code listing (A) – blocking methodology

File read / write

```
AssignFile(F, 'c:\testfile.dat');
Reset(F);
Readln(F, S);
S := FormatDateTime('dd mmm yyyy',Time)
             + #32 + S;
WriteLn(F,S);
CloseFile(F);
```

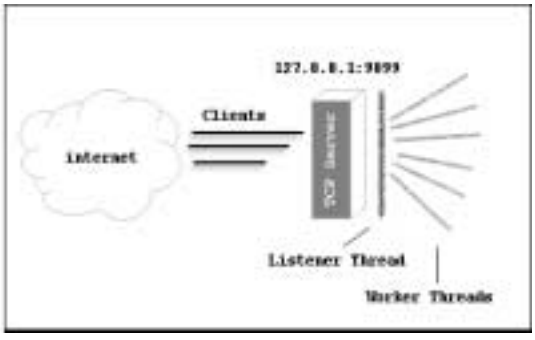
Indy socket

```
with IdTCPClient1 do
  begin
  Connect;
  S := ReadLn;
  S := FormatDateTime('dd mmm yyyy',
            Time) + #32 + S;
  WriteLn(S);
  Disconnect;
  end;
```

Threading model

One of the core top level classes in Indy is the TidTCPServer (the *id*, by the way, standing for Indy!). TCPServer acts as the main sentry *(or* entrypoint*)* for every client that wishes to interact with your service. Once it has checked the client at the gate, it calls a subordinate thread to take the client off privately and deal with it, leaving it free to handle new requests,as illustrated in diagram(B).

diagram (B) – Indy threading model



In diagram (B), the TCP server is running on an IP address of 127.0.0.1 (Localhost), listening on port 9099. Each time a client makes a new connection to the server, it first encounters the Listener thread, which offloads it to a newly created worker thread. It is the responsibility of the worker thread to handle all communication with the client. It may seem a complex way of handling things, but it is both extremely powerful and efficient. Later articles and classes will explain such concepts as thread pooling, and thread reuse - critical processes for load balancing heavily utilised servers effectively.

There are three main core events in the lifetime of a client thread connection to a TCPServer worker thread:

- OnConnect
- OnExecute
- OnDisconnect

The first and last events are self-explanatory. The main work of communication, however, is carried out in the OnExecute event. Worker threads in Indy are referred to as PEER THREADS. Each event gives us access to the peer thread usually in the following manner:

```
IdTCPServer1Execute(AThread:
                TIdPeerThread);
```

The main property of this peer thread we are concerned with is the actual connection to the client.

```
AThread.connection
```

The server peer thread operates in a similar fashion as that described for the client in code listing (A), in that it operates in a blocking manner:

```
S := AThread.connection.ReadLn;
AThread.connection.WriteLn(FormatDateTime(
                    'dd mmm yyyy',Time));
```

The conversation between client and server could be seen as follows:

Client asks a question:

```
IdTCPClient1 do
  begin
  Connect;
  WriteLn('Time?');
  S := ReadLn;
```

Server answers:

```
S := AThread.connection.Readln;
If s = 'Time?' then
   Athread.connection.WriteLn(
      FormatDateTime('hh:nn:ss',Time));
```

In addition to the text based ReadLn/WriteLn commands, other basic and frequently functions are:

- ReadInteger
- ReadBuffer
- ReadStream

(and corresponding writes)

Protocols

Simple as it seems, the example of the conversation between client and server in the previous section can be termed a protocol. A protocol is a formal method of communication between the client and the server. If the client for example wrote 'What Time?' the server would not understand it because the protocol demands that the command is simply 'Time?' – nothing more, nothing less.

Electronic protocols are all around us, and we use them every day without realising it. For example, when we send an email, we are most likely using the SMTP protocol (Internet RFC 821). Simple Mail Transfer Protocol exists so that mailservers of different makes and on different platforms can communicate with other servers and indeed clients, using a simple and common language.

When sending an e-mail message from your favourite mail client, your client is actually having the following conversation with the server:

```
Client : MAIL FROM: MyName@MyDomain.com
Server : 250 OK
Client : RCPT TO: MyFriend@ADomain.com
Server : 250 OK
Client : DATA :
       {content of the message}
Client : <crlf>.<crlf>
       {above signifies end of message}
Server : 250 OK
```

If the server encounters an error, it will send back an error code and (usually) a plain text description of the code. Using Internet standard protocols, almost every higher-level component in Indy has been written based on the standard TCPClient and TCPServer.

Most of the Internet protocols that we use and develop custom solutions for on a daily basis (e.g.: POP3, SMTP, HTTP) are provided in Indy as a matter of course. The classes I find myself making most use of are the higher, more abstract classes that are, in fact, the ancestors of most of the well-known protocols in any case.

The reason I gave a brief introduction to protocols above is that almost everything you do in sockets either requires you to use an existing protocol provided by someone else, or to make up your own protocol. There are over 70 demos provided in Indy, some ten of these alone use roll-your-own protocols to accomplish some task or other. I would strongly encourage you to take some time and investigate these demos before shouting for help on the newsgroups. In most cases, the answers to frequently asked questions are to be found clearly in the code from the demonstration programs provided with Indy.

Next please!

In upcoming articles and sessions, Hadi Hariri and I intend to go in-depth into Indy, demonstrating its power for use with encryption, nTier data distribution, custom mailers, and more. We will also cover some of the very powerful concepts in Indy such as Commandhandlers, IOHandlers, etc. If you have specific questions that are not covered in the demonstration programs, feel free to contact any of the Indy team through the Borland newsgroups. For obvious reasons, technical questions directed towards our private email addresses are redirected to newsgroups.

Newsgroups:

- borland.public.delphi.internet.winsock
- borland.public.cppbuilder.internet

Indy website:

- http://www.nevrona.com/indy

Useful articles:

- http://www.nevrona.com/indy/Articles.html

Internet RFC/STD/FYI/BCP Archives

- http://www.landfield.com/rfcs/index.html

Books

- Building Kylix Applications (ISBN 007212947-6) - *Cary Jensen*. Indy Chapters by *Chad Z. Hower.*
- Delphi Developer's Guide to Internet Direct - *Chad Z. Hower, Allen O'Neill, Hadi Hariri...* Release Q2 2002

Speaking engagements

- Dublin, November 23rd – Allen O'Neill @ UK-BUG meeting.
- London, March 5th – Allen O'Neill and Hadi Hariri – UK-BUG Masterclass.

About the author

Allen O'Neill works for Springboard Technologies Limited in Dublin (European capital of weekend stag parties!). He is heavily involved in the OpenSource community through Indy, is an active speaker at various technical gatherings and is currently co-authoring a book on Internet Direct with Hadi Hariri and Chad Z. Hower (Wordware publishers, release due Q2/200/2). Allen has broad technical expertise, covering Internet, sockets, database, systems architecture, project management and OOP. Allen is married, has two large dogs and a passion for fast motorcycles. He may be reached at allen_oneill@hotmail.com

Pictured from left to right: Allen O'Neill (Ireland), Hadi Hariri (Spain), Chad Hower (USA).