Indy in Depth

Indy

Taming Internet development one protocol at a time.

Indy in Depth

Acceptable Use - Personal

If a personal license has been purchased, this document, its contents, and all associated sample code is licensed only for use by the individual that it was licensed to. It may not be shared or distributed to other individuals.

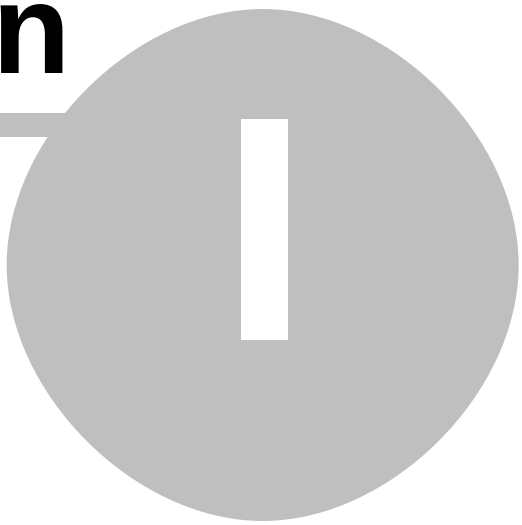
Acceptable Use - Corporate

If a corporate license has been purchased, this document, its contents, and all associated sample code is licensed only for use by the corporation, its employees, and its current contractors.

The corporation may redistribute this document and assocated source code if access is restricted to only employees and current contractors of the corporation.

Table of Contents

Section



Introduction

1 Introduction

Welcome to Indy in Depth.

Authors:

- **Chad Z. Hower a.k.a Kudzu** - Original Indy Author and current Indy Project Coordinator
- **Hadi Hariri** - Indy Project Co-Coordinator

This material is based on material developed and presented at developer conferences in nearly a dozen countries worldwide and material developed and published in developer magazines.

1.1 About the Book

Please note that this is a preview release! There is a lot of material that is being worked. This includes consolidation, updating, and importing from other sources. In addition you will see several topics that are filled with notes and babble. That is where we are working and it gives you an insight into what is coming. Do not worry if some sections do not make sense at this time.

We expect to release updates about once per month and for the first few months a lot of new material will be appearing.

Thanks for your support and patience!

1.1.1 Feedback

Please direct any feedback related to this book to indy@atozedsoftware.com. Do not send technical support requests to this e-mail. For technical support please see the technical support section .

1.1.2 Updates

Updates of Indy in Depth can be found at http://www.atozedsoftware.com/.

1.1.3 Demos

All demos referenced in this book are available at the Indy Demo Playground at www.atozedsoftware.com.

1.2 Legal

1.2.1 Copyright

1.2.2 Acceptable Use - Personal

1.2.3 Acceptable Use - Corporate

1.3 Additional Information

1.3.1 Other Resources

- **Main Indy Website** - http://www.indyproject.org/
- **Worldwide Indy Mirrors** - http://indy.torry.net/
- **Indy Portal** - Subscribe to Indy News, Read Free Articles, Download Extra Demos. http://www.atozedsoftware.com/indy/
- **Kudzu World** - Chad Z. Hower's strange corner of the WWW. http://www.hower.org/Kudzu/

1.3.2 Further Reading

The following books contain chapters about Indy.

- **Building Kylix Applications** by Cary Jensen
- **Mastering Delphi** by Marco Cantu
- **SDGN Magazine** - SDGN Magazine, the official magazine of SDGN has a regular column called The *Indy Pit Stop* written by Chad Z. Hower a.k.a Kudzu.

1.4 Credits

Thanks to the following people, groups, companies:

- The **Indy Pit Crew and the Mercury Team** for their long hours and dedication to Indy.
- Lest not, the **many others** who have assisted with Indy that we have forgotten.
- **ECSoftware** for producing Help and Manual with which this document is produced and written. http://www.helpandmanual.com

Section II

Technical Support

2 Technical Support

2.1 Note

Please do NOT send email directly to the members of the Indy Pit Crew unless specifically requested to do so (typically when a crew member needs additional information) or responding to an email from a member of the Indy Pit Crew.

Due to the amount of time the members of the Indy Pit Crew spend in the newsgroups while still earning their daily bread, they are unable to respond to technical questions via email. Technical questions are best directed to the appropriate newsgroup.

If you require fast priority answers beyond what we provide freely out of our spare time and of our own free will with no financial compensation, please see the paid support options below. Even with paid support, since Indy itself is completely free, it is still a cheap option.

If you are still not convinced why you should not send us mail directly please consider the following additional items:

- **Indy is done by a team.** This means that certain people are responsible for certain parts of code. By sending email, you likely have sent it to the wrong person who is not the one who knows the answer quickly.
- **Email reaches one person** and thus puts the load completely on that person. By using the options below, the complete Indy Pit Crew as well as many other Indy users will see your question and the appropriate person will be able to respond. In addition you are likely to receive a variety of answers from different people to further assist you.
- **Also please read** How To Ask Questions The Smart Way

2.2 Free Support

Free support can be obtained by visiting the Borland Newsgroups

- borland.public.delphi.internet.winsock
- borland.public.cppbuilder.internet.socket
- borland.public.kylix.internet.sockets

All of these newsgroups exist at newsgroups.borland.com. If you cannot access these via NNTP, you can use your webrower to use the web interface at http://newsgroups.borland.com. Team Indy monitors these groups regularly, and everyone will benefit from your post.

2.3 Commercial Priority Support

Open source software is fantastic, but often lacks support. Indy Experts Support solves this problem for Indy. Indy Experts Support provides your company with priority e-mail and phone support in a very cost effective manner.

Indy Experts Support includes:

- Priority e-mail support
- Phone support
- Direct access to Indy authors and project leaders
- Custom code snippets and small projects

- Priority bug fixes and Code review.
- Consulting via Indy Consulting

More detailed information about Indy Experts support can be found at http://www.atozedsoftware.com/indy/support/ .

The Indy Experts Support team consists of the foremost experts in Indy:

- **Chad Z. Hower** - Original Author and Project Leader of Indy
- **Hadi Hariri** - Project Co-Leader of Indy
- **Doychin Bondzhev** - Prominent Indy Core Team Member

2.4 SSL Support

Intellicom hosts a SSL support forum .

2.5 Bug Reports

All bugs should be reported using the bug reporting form on the Indy website at http://www.nevrona.com/indy or http://indy.torry.net/

2.6 Winsock 2

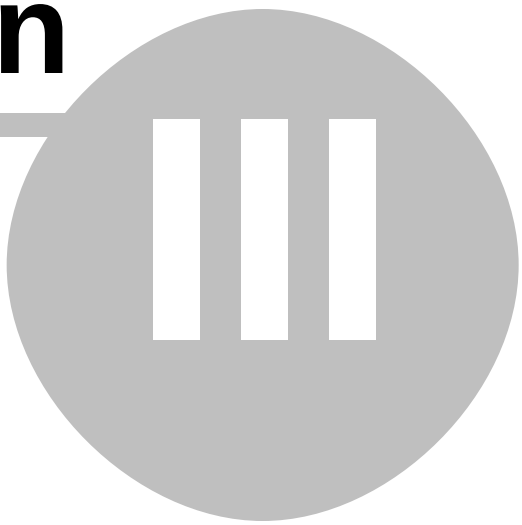
Indy 9.0 requires Winsock 2.0. Windows 98, Windows NT and Windows 2000 include Winsock 2.0 by default. When Indy is initialized, it checks the Winsock version.

Windows 95 is the only operating system that does not support Winsock 2.0 by default. However, it can be upgraded to Winsock 2.0 by downloading a small update from Microsoft. The URL for this download is:

http://www.microsoft.com/windows95/downloads/contents/wuadmintools/s_wunetworkingtools/w95sockets2/

The patch should be applied in any case because the Winsock that ships with Windows 95 contains several serious bugs and memory leaks that this patch also fixes. Windows 95 currently accounts for less than 5% of current installed systems. Very few of those are connected to a network or the Internet, so this will likely have very little impact on your applications deployment.

Section



Introduction to Sockets

3 Introduction to Sockets

3.1 Overview

This article is an introduction to socket (TCP/IP sockets) concepts. It as not meant to be a complete coverage of all socket topics; it is meant as a primer to educate the reader at a level at which socket programming can be easily communicated.

There are several concepts that must be introduced first. Whenever possible, concepts will be likened to a real world concept you are likely familiar with: a phone system.

3.2 TCP/IP

TCP/IP is short for *Transmission Control Protocol and Internet Protocol*.

TCP/IP can mean many things. It is often used very generically as a "catch all" word. In most cases, it refers to the network protocol itself.

3.3 Client

A client is a process that initiates a connection. Typically, clients talk to one server at a time. If a process needs to talk to multiple servers, it creates multiple clients.

Likening it to a phone call, a client would be the person who makes a call.

3.4 Server

A server is a process that answers incoming requests. A typical server handles numerous requests from many clients simultaneously. Each connection from the server to the client, however, is a separate socket.

Likening it to a phone call, the server would be the person (or voice mail, interactive system, etc.) who answers the phone when it rings. A server is typically set up so that it can handle multiple incoming phone calls. This is similar to how a call center might handle many calls by having hundreds of operators and routing each incoming call to an available operator.

3.5 IP Address

Each computer on a TCP/IP network has a unique address associated with it. Some computers may have more than one address associated with them. An IP address is a 32-bit number and is usually represented in dot notation, e.g. 192.168.0.1. Each section represents one byte of the 32-bit address.

An IP address is like a phone number. However just as a location (residence, business, etc.) can have one or more phone number, an individual machine may have more than one IP address assigned to it. Machines that have more than one IP address are said to be multi-homed.

To talk to someone at a specific location, a connection attempt is initiated (a call is placed and the dialed location's phone rings) by dialing the phone number for that location. The party at the ringing end of the phone can then decide whether or not to answer the phone.

Often an IP address is referred to simply as an "IP" for short.

3.6 Port

A port is an integer number that identifies which application or service the client wishes to contact at a given IP address.

A port is much like a phone extension. Dialing a phone number will get you to a location, but with TCP/IP, every location also has an extension. There is no default extension, as with a residential phone, thus in addition to an IP address, a port must always be specified when connecting to a server.

When an application server is ready to accept incoming requests, it begins to listen on a port. This is why sometimes application or protocol is used interchangeably with the word port. When a client wants to talk to a server, it must know where the application is (IP address/phone number), and which port (extension) the application is listening (answering) on.

Typically, applications have a fixed port so that no matter what machine they run on, the port is fixed for that type of application. For example, HTTP (Web) uses port 80, and FTP uses port 21. When you want to retrieve a Web page, you only need to know the location of the computer you wish to retrieve it from, as you know HTTP uses port 80.

Port numbers below 1024 are reserved, and should only be used if you are talking to or implementing a known protocol that has such a port reserved for its use. Most popular protocols use reserved port numbers.

3.7 Protocol

The word *protocol* is derived from the Greek protocollon. A protocollon was a leaf of paper glued to a manuscript volume, describing its contents.

In terms of TCP/IP, a protocol is a description of how something works. But more generically it is generally referring to one of two things:

1. The type of socket.
2. A higher level command protocol.

When referring to sockets, the protocol specifies what kind of socket it is. Common types of sockets are TCP, UDP and ICMP sockets.

When referring to a higher level protocol, it refers to the commands and responses used to perform the function that is desired. These protocols are described in RFC's. Examples of such protocols are HTTP, FTP, and SMTP.

3.8 Socket

All references to a socket in this article are references to TCP/IP. A socket is the combination of an IP address, a port number, and a protocol. A socket is also a virtual communication conduit between two processes. These processes may be locally (residing on the same computer), or remotely.

A socket is like a phone connection that carries a conversation. To have a conversation, you must first make the call, and have the other party answer; otherwise, no connection (socket) will be established.

3.9 Host Name

Host names are "human-readable" names for IP addresses. An example host name is www.nevrona.com. Every host name has an equivalent IP address. For example, www.nevrona.com resolves to 208.225.207.130.

Host names are used both to make it easier on us humans, and to allow a computer to change its IP address without causing all of its potential clients (callers) to lose track of it.

A host name is like a person's name or a business name. A person or business can change their phone number, but we can still contact them.

3.10 DNS

DNS is short for *Domain Name Service.*

DNS is the service that translates host names into IP addresses. To establish a connection, an IP address must be used, so DNS is used to look up the IP address first.

To make a phone call, you must dial by using the phone number. You cannot dial using a person's name. If you do not have the person's phone number, or it has changed, you would look up the person's phone number in the phone book, or call directory assistance. Thus, DNS is the phone book/directory assistance for the Internet.

3.11 TCP

TCP is short for *Transmission Control Protocol*.

TCP is sometimes also referred to as stream protocol. TCP/IP includes many protocols and many ways to communicate. The most common transports are TCP and UDP. TCP is a connection-based protocol - that is, you must connect to a server before you can send data. TCP guarantees delivery and accuracy of the data sent and received on the connection. TCP also guarantees that data will arrive in the order that it was sent. Most things that use TCP/IP use TCP for their transport.

TCP connections are like using a phone call to carry on a conversation.

3.12 UDP

UDP is short for *User Datagram Protocol*.

UDP is for datagrams and is connectionless. UDP allows "lightweight" packets to be sent to a host without having to first connect to another host. UDP packets are not guaranteed to arrive at their destination, and may not arrive in the same order they were sent. When sending a UDP packet, it is sent in one block. Therefore, you must not exceed the maximum packet size specified by your TCP/IP stack.

Because of these factors, many people assume UDP is nearly useless. This is not the case. Many streaming protocols, such as RealAudio, use UDP.

Note: The term "streaming" can be easily confused with "stream" connection, which is TCP. When you see these terms, you need to determine the context in which they are used to determine their proper meaning.

The reliability of UDP packets depends on the reliability and the saturation of the network. UDP packets are also often used on applications that run on a LAN, as the LAN is very reliable. UDP packets across the Internet are generally reliable as well and can be used with error correction or more often interpolation. Delivery however cannot be guaranteed on any network - so do not assume your data will always arrive at your destination.

Because UDP does not have delivery confirmation, its not guaranteed to arrive. If you send a UDP packet to another host, you have no way of knowing if it actually arrived at its destination. The stack will not - and cannot - determine this, and thus will not provide an error if the packet did not reach its destination. If you need this information, you need to send some sort of return notification back from the remote host.

UDP is like sending someone a message on a traditional pager. You know you sent it, but you do not know if they received it. The pager may not exist, may be out of the service area, may not be on, or may not be functioning. In addition, the pager network may lose the page. Unless the person pages you back, you do not know if your message was delivered. In addition, if you send multiple pages, it is possible for them to arrive out of order.

More Information on UDP in the UDP Chapter

3.13 ICMP

ICMP is short for *Internet Control Message Protocol*.

ICMP is a control and maintenance protocol. Typically, you will not need to use ICMP. Typically, it is used to communicate with routers and other network devices. ICMP allows nodes to share IP status and error information. ICMP is used for PING, TRACEROUTE, and other such protocols.

3.14 HOSTS

HOSTS is a text file that contains a local host lookup table. When the stack attempts to resolve a host name to an IP address, it firsts look in the HOSTS file. If a matching entry exists, it will use that entry. If an entry does not exist, it will proceed to use DNS.

Here is an example HOSTS file:

```
# This is a sample HOSTS file
caesar     192.168.0.4  # Server computer
augustus   192.168.0.5  # Firewall computer
```

The host name and IP address can be separated by spaces or the tab character. Comments can also be inserted into the file by preceding the line with a # character.

HOSTS can be used to fake entries, or override DNS entries. The HOSTS file is often used on computers on a small LAN that have no DNS server. The HOSTS file is also useful for overriding host IP addresses for debugging. You do not need to read the HOSTS file; the stack will take care of this detail for you transparently whenever name resolution is necessary.

3.15 SERVICES

A SERVICES file is similar to a HOSTS file. Instead of resolving host names into IP addresses, it resolves service names into the port numbers they are assigned.

The following is a partial SERVICES file. You can look on your computer to see a complete file, or

obtain RFC 1700. RFC 1700 contains assigned and reserved port numbers:

```
echo                 7/tcp
echo                 7/udp
discard              9/tcp    sink null
discard              9/udp    sink null
systat              11/tcp    users                    #Active users
systat              11/tcp    users                    #Active users
daytime             13/tcp
daytime             13/udp
qotd                17/tcp    quote                    #Quote of the day
qotd                17/udp    quote                    #Quote of the day
chargen             19/tcp    ttytst source            #Character
                                                       # generator
chargen             19/udp    ttytst source            #Character
                                                       # generator
ftp-data            20/tcp                             #FTP data
ftp                 21/tcp                             #FTP control
telnet              23/tcp
smtp                25/tcp    mail                     #Simple Mail
                                                       # Transfer Protocol
```

The format of each entry is:

```
<service name> <port number>/<protocol> [aliases...] [#<comment>]
```

You do not need to read the SERVICES file; the stack will also take care of this detail for you. The SERVICES file is read by certain function calls in the stack; however, most programs do not call these functions, and therefore ignore its values. For example, many FTP programs default to 21 without ever using the stack to look up the port for the 'ftp' entry.

Normally, you should never modify this file. Some programs, however, add an entry to this file and actually use it. You can then change this entry to tell those programs to use another port. One such program is Interbase. Interbase makes the following entry:

```
gds_db            3050/tcp
```

You can change this entry to make Interbase use a different port. While it is not common practice to do this, it is a good practice. It should be considered if you write socket applications, especially servers. It is also good practice for clients to use the stack to look up the value in SERVICES, especially for non-standard protocols. If no entry is found, a default should be used.

3.16 Localhost (Loopback)

LOCALHOST is similar to "Self" in Delphi, or "this" in C++. LOCALHOST refers to the computer that the application is executing on. It is a loop back address, and has a physical IP address of 127.0.0.1. If 127.0.0.1 is used in a client, it always loops back and looks for a server on the same computer as the client.

This is useful for debugging. It can also be used to contact any service running on your computer. If you have a local web server, instead of needing to know the IP address of the computer or have each developer change it in test scripts, you can specify 127.0.0.1.

3.17 Ping

Ping is a protocol that verifies whether a host is reachable by the local computer. Ping is usually used in a diagnostic capacity.

Ping is available as a command line program. Its usage is:

```
ping <host name or IP>
```

The following is sample output of a successful Ping:

If a host cannot be reached, the output will look similar to this:

```
C:\>ping 192.168.0.200

Pinging 192.168.0.200 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.200:
    Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),
```

3.18 TraceRoute

TCP/IP packets do not travel directly from one host to another. Packets are routed much like a car travels from one house to another. Typically, the car must travel on more than one road to reach its destination. TCP/IP packets travel much in the same way. Each time a packet changes "roads" at an "interchange," it travels through a node. By obtaining a list of nodes or "interchanges" that a packet must travel through between hosts, you can determine its path. This is quite useful in determining why a host cannot be reached.

Traceroute exists as a command line utility. Traceroute displays a list of IP routers (nodes) that are used in delivering a packet from your computer to the specified host, and how long each trip (hop) took. These times can be useful in determining bottlenecks. Traceroute can also display the last router that successfully handled your packet in the case of a failed transfer. Traceroute is used to further diagnose problems detected with Ping.

The following is sample output of a successful Traceroute:

```
C:\>tracert www.atozedsoftware.com

Tracing route to www.atozedsoftware.com [213.239.44.103]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  server.mshome.net [192.168.0.1]
  2    54 ms    54 ms    50 ms  102.111.0.13
  3    54 ms    51 ms    53 ms  192.168.0.9
  4    55 ms    54 ms    54 ms  192.168.5.2
  5    55 ms   232 ms    53 ms  195.14.128.42
  6    56 ms    55 ms    54 ms  cosmos-e.cytanet.net [195.14.157.1]
  7   239 ms   237 ms   237 ms  ds3-6-0-cr02.nyc01.pccwbtn.net [63.218.9.1]
  8   304 ms   304 ms   303 ms  ge-4-2-cr02.ldn01.pccwbtn.net [63.218.12.66]
  9   304 ms   307 ms   307 ms  linx.uk2net.com [195.66.224.19]
 10   309 ms   302 ms   306 ms  gw12k-hex.uk2net.com [213.239.57.1]
 11   307 ms   306 ms   305 ms  pop3 [213.239.44.103]

Trace complete.
```

3.19 LAN

LAN is short for *Local Area Network*.

What defines a LAN is a fuzzy term, and can vary depending on each network configuration. However a LAN normally refers to all systems connected by Ethernet (or in some cases Token ring or other), hubs and switches. The LAN does not include other LAN's that are hooked together using bridges or routers. So the LAN consists of only places that network traffic can reach without needing to be passed to another network by a bridge or router.

Think of a LAN like the surface streets of a city, with bridges and routers being the highways that connect cities together.

3.20 WAN

WAN is short for *Wide Area Network*.

A WAN collectively refers to many LAN's hooked together using bridges and routers to form a larger network.

Using the city example again, a WAN would consist of many cities (LAN's) connected together by highways and interstates. The Internet itself is also classified as a WAN.

3.21 IETF

The IETF (Internet Engineering Task Force) is an open community that promotes the operation, stability, and evolution of the Internet. The IETF works much like Open Source software development teams. The IETF can be found at http://www.ietf.org/.

3.22 RFC

RFC is short for *Request for Comments*.

RFC's are the official documents of the IETF that describe and detail protocols of the Internet. RFC's are identified by number such as RFC 822.

There are many mirrors that contain RFC documents on the Internet. A good one that allows searching is available at:
http://www.rfc-editor.org/

RFC Editor (The website listed above) describes RFCs as below:

> *The Requests for Comments (RFC) document series is a set of technical and organizational notes about the Internet (orginally the ARPANET), beginning in 1969. Memos in the RFC series discuss many aspects of computer networking, including protocols, procedures, programs, and concepts, as well as meeting notes, opinions, and sometimes humor.*

3.23 Thread

A thread is an execution path of a program. Many programs only have one thread. However other threads can be created to create sperate execution paths.

In systems with multiple CPU's the threads can be distributed among different CPU's for faster execution.

In systems with only one CPU, multiple threads can still be executed by using preemption. Preemption multi-tasks the CPU into processing many threads by giving each one a small time slice. The threads are therefore virtually executed as if there were individual CPU's executing each thread in parallel.

3.24 Fork

Unix until recently did not support threading. Instead of threading, Unix used forking. With threading, a separate line of execution is established, but it exists in the same process as other threads, and thus shares the same memory space. Forking causes a process itself to be "split". A new process is started and handles passed to it.

Forking is not as efficient as threading, but does have some advantages. Forking is more stable. Forking is also easier to program in many cases.

Forking is very common in Unix as the kernel uses and supports it , while threading is quite new.

3.25 Winsock

Winsock is short for *Windows Sockets*.

Winsock is a defined and documented standard API for programming network protocols. Most commonly it is used to program TCP/IP, but can also be used to program Novell (IPX/SPX) and other network protocols. Winsock is accessible as a DLL that is part of Win32.

3.26 Stack

The term *Stack* refers to the layer of the operating system which implements the network and provides the API for developers to access the network.

In Windows the stack is implemented as Winsock.

3.27 Network Byte Order

Different computer systems store numeric data in different orders. Some computers store numbers with the least significant byte first (LSB) while others store the most significant byte first (MSB). When a network involved, it is not always known what order the computer on the other side of a connection is using. To solve this problem there is a standard order for numbers to be stored in and transmitted on the network called Network Byte Order. Network Byte Order is a fixed byte order that applications should use when transmitting binary numbers.

Section

Introduction to Indy

4 Introduction to Indy

4.1 The Indy Way

Indy is designed from the ground up to be threadable. Building servers and clients in Indy is similar to the way Unix servers and clients are built, except that it is much easier, because you have Indy and Delphi. Unix applications typically call the stack directly with little or no abstraction layer.

Typically Unix servers have one or more listener processes which look for incoming client requests. For each client that it needs to serve, it will fork a new process to handle each client. This make programming very easy as each process deals with only one client. The process also runs in its own security context, which can be set by the listener or the process based on credentials, authentication, or other means.

Indy servers work in a very similar manner. Windows unlike Unix does not fork well, but it does thread well. Indy servers allocate a thread for each client connection.

Indy servers set up a listening thread that is separate from the main thread of the program. The listener thread listens for incoming client requests. For each client that it answers, it spawns a new thread to service that client. The appropriate events are then fired within the context of that thread.

4.2 Indy Methodology

Indy is different from other socket components with which you may be familiar. If you have never worked with other socket components, you will find Indy very easy as Indy operates in a fashion that you expect. If you have worked with other socket components, please forget what you know. It will only hinder you and cause you to make false assumptions.

Nearly all other components use non-blocking calls and act asynchronously. They require you to respond to events, set up state machines, and often perform wait loops.

For example, with other components, when you call connect you must wait for a connect event to fire, or loop until a property indicates that you are connected. With Indy, you merely call Connect, and wait for it to return. If it succeeds, it will return when it has completed its task. If it fails, it will raise an exception.

Working with Indy is very much like working with files. Indy allows you to put all your code in one place, instead of scattered throughout different events. In addition, most find that Indy is much easier to use. Indy is also designed to work well with threads. If at anytime you have trouble implementing something with Indy, step back and approach it like a file.

4.3 How Indy is Different

- Indy uses the blocking socket API.
- Indy does not rely on events. Indy has events that can be used for informational purposes, but are not required.
- Indy is designed for threads. Indy however can be used without threads.
- Indy is programmed using sequential programming.
- Indy has a high level of abstraction. Most socket components do not effectively isolate the programmer from stack. Most socket components instead of isolating the user from the complexities of stack merely pass them on with a Delphi / C++ Builder wrapper.

4.4 Overview of Clients

Indy is designed to provide a very high level of abstraction. Intricacies and details of the TCP/IP stack are hidden from the Indy programmer.

A typical Indy client session looks like this:

```
with IndyClient do begin
  Host := 'postcodes.atozedsoftware.com'; // Host to call
  Port := 6000; // Port to call the server on
  Connect; Try
    // Do your communication
  finally Disconnect; end;
end;
```

4.5 Overview of Servers

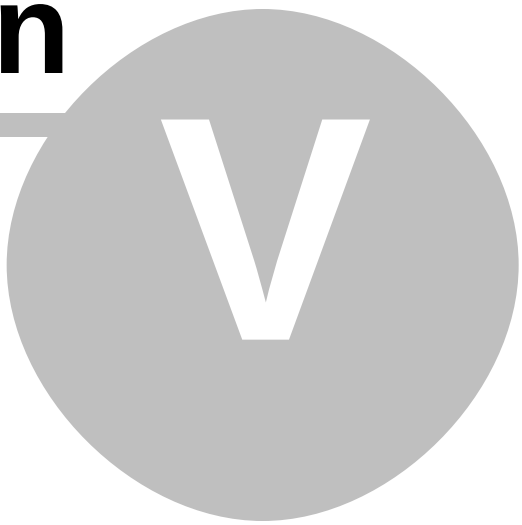
Indy server components create a listener thread that is separate from the main thread of the program. The listener thread listens for incoming client requests. For each client that it answers, it then spawns a new thread to service that client. The appropriate events are then fired within the context of that thread.

4.6 Threading

Threading is the process of using threads to implement functionality. Indy uses threads extensively in its server implementations and threading is useful for clients as well.

Non-blocking sockets can be threaded as well, but they require some extra handling and their advantages are lost with blocking sockets.

Section



Blocking vs Non-Blocking

5 Blocking vs Non-Blocking

5.1 Programming Models

There are two programming models for programming sockets under Windows, blocking and non-blocking. Sometimes they are also called synchronous (blocking) and asynchronous (non-blocking). Throughout this document the terms blocking and non-blocking will be used.

Under Unix, only blocking is supported.

5.2 More Models

There are actually a few other models that have been implemented as well. These include completion ports, and overlapped I/O. However usage of these requires significantly more code and is typically reserved for advanced server applications.

In addition, these models are not cross platform and are implemented differently and to different degrees among different operating systems.

Indy 10 contains support for these additional models.

5.3 Blocking

Indy uses blocking socket calls. Blocking calls are much like reading and writing to a file. When you read data, or write data, the function will not return until the operation is complete. The difference from working with files versus sockets is that operations may take much longer because data may not be immediately ready for reading or writing. A read or write operation can only operate as fast as the network or the modem can receive or transmit data.

With Indy, to connect a socket one simply calls the Connect method and waits for it to return. If the Connect method succeeds, it will return when it connect. If the Connect method fails, it will raise the appropriate exception.

5.4 Non-Blocking

Non-blocking sockets work on a system of events. Calls are made, and when they complete or need attention, an event is fired.

For example, when attempting to connect a socket, you must call the Connect method. The Connect method will return immediately, before the socket is connected. When the socket is connected, an event will occur. This requires that the communication logic be split up into many procedures, or the use of polling loops.

5.5 History of Winsock

In the beginning, there was Unix. In this case, Berkely Unix. Berkely Unix had a standard socket API that became adopted and spread among the Unix flavors.

Then there was Windows, and eventually someone decided it would be a good idea to have the ability to program TCP/IP on Windows. So, they ported the popular Unix socket API. This allowed for much of the existing Unix code to be ported to Windows easily.

5.6 Blocking is not Evil

Blocking sockets have been repeatedly attacked based on false premises. Contrary to popular belief, blocking sockets are not evil.

When the Unix socket API was ported to Windows, it was named Winsock. Winsock is short for "Windows Sockets". During the port, a problem quickly arose. In Unix, it was common to fork. Forking is similar to multi threading, but with separate processes instead of threads. Unix clients and servers fork processes for each socket. These processes then execute independently and use blocking sockets.

Windows 3.1 could not fork effectively because it lacked preemptive multitasking and other necessary support. Windows 3.1 also did not have support for threads. Using blocking sockets caused user interfaces to freeze. Because Windows 3.1 used cooperative multi tasking, this also caused all programs on the system to become unresponsive as well. Because this was highly undesirable, non-blocking extensions were added to WinSock to allow Windows 3.x with its shortcomings to use Winsock without freezing the entire system. These extensions however required a radically different way of programming sockets. Microsoft and others found it in their best interests to vilify blocking sockets rather than highlight the shortcomings of Windows 3.1.

With the introduction of Windows NT and Windows 95, Windows gained the ability to preemptively multitask and thread. By that time, most programmers had been convinced that blocking sockets were evil. Bridges had been burned, and so the vilification of blocking sockets continued.

In reality, the blocking API is the ONLY API that Unix supports.

Some extensions have been added for non-blocking sockets in Unix. These extensions however work quite differently than the non-blocking extensions in Windows. The extensions are not standard among all Unix platforms, and not in wide use. Blocking sockets in Unix are still used in almost every instance, and this usage will continue to be so.

Blocking sockets also offer other advantages. Blocking sockets are much better for threading, security, and other aspects.

5.7 Pros of Blocking

1. **Easy to program** - Blocking sockets are very easy to program. All user code can exist in one area, and in a sequential order.
2. **Cross Platform** - Since Unix uses blocking sockets, portable code can be written easily. Indy uses this fact to achieve its single source cross platform ability. Other socket components that are cross platform simulate non-blocking behavior by using blocking calls internally.
3. **Work well in threads** - Since usage of blocking sockets is sequential in nature, they are inherently encapsulated and therefore well suited for use in threads.
4. **Do not depend on messages** - Non-blocking sockets depend on the window messaging system. When used in threads separate message queues must be created. When not used in threads, bottleneck situations easily occur when multiple connections are handled.

5.8 Cons of Blocking

1. **User Interface "Freeze" with clients** - Blocking socket calls do not return until they have accomplished their task. When such calls are made in the main thread of an application, the application cannot process the user interface messages. This causes the User Interface to "freeze". The freeze is caused because the update, repaint and other messages cannot be processed until the blocking socket calls return control to the applications message handler.

5.9 TIdAntiFreeze

Indy has a special component that solves the user interface freeze problem transparently. The existence of a single TIdAntiFreeze instance in an application, allows the use of blocking calls in the main thread without the user interface being frozen. TIdAntiFreeze is covered more in depth later.

Use of a TIdAntiFreeze allows for all the advantages of blocking sockets, without the prominent disadvantage.

5.10 Pros of Non-Blocking

1. **No User Interface "freeze" with clients** - Since the user code responds to events windows has control between socket events. Because of this, Windows can also respond to other windows messages.
2. **Can multitask without threading** - A single thread can handle many sockets.
3. **Lighter weight with many sockets** - Since many sockets can be handled without the need for threads, the overall memory and CPU requirements are usually less.

5.11 Cons of Non-Blocking

1. **More difficult to program** - Non-blocking requires the use of polling or events. Events are the more common method, as polling loops are quite inefficient. The use of events requires that the user code be split into many procedures thus state tracking added. This leads to brittle code that is prone to bugs and difficult to modify.

5.12 Comparison

If you are familiar with Indy and its methodologies you may wish to skip ahead. If you have however programmed sockets before using Indy, this section will be useful to you.

For those who have never programmed sockets before Indy is very natural and easy to use. But for those who have programmed sockets before, Indy typically presents an initial stumbling block. This is because Indy works differently. This does not mean that other approaches are wrong, its just that Indy is different. Trying to program Indy in the same manner as other socket libraries is like trying to cook in a microwave the same way that you cook on the stove. The results will be disastrous and maybe even explosive.

If you have used other socket libraries before, please take this simple and subtle advice:

Forget what you know!

This is much easier said then done however as old habits die hard. To emphasize the difference an abstracted comparison will be presented using an example. To abstract the concepts and disconnect your thought patterns from sockets, files will be used instead. This document assumes that you know how to read and write to files. (That is to say, no Visual Basic programmers are reading this document).

5.13 Files vs. Sockets

The major difference between files and sockets is that typically file access is rather fast. However file access is not always fast. Floppy disk, network drives, backup storage and hierarchical storage management systems often provide for slow response times.

5.14 Scenario

A simple scenario will be presented that will simply write data to a file. While the procedure to do this is quite simple, here it is detailed for demonstration purposes:

1. Open the file
2. Write the data
3. Close the file

5.15 Blocking File Write

A blocking file write would look something like this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Indy Rules the (Kudzu) World !' + #13#10;
  try
    // Open the file
    with TFileStream.Create( 'c:\temp\test.dat', fmCreate) do try
      // Write data to the file
      WriteBuffer(s[1], Length(s));
    // Close the file
    finally Free; end;
  end;
end;
```

As you can see, it matches one for one for the steps outlined previously and as marked by the comments. The code is sequential and readily understood.

5.16 Non-Blocking File Write

There is no such thing as a non-blocking file write (except maybe overlapped I/O, but that is beyond the scope document and the code is even uglier), but if there was it would look something like as follows. File1 is an imaginary non-blocking file component which has been dropped on the form.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  File1.Filename := 'd:\temp\test.dat';
  File1.Open;
end;

procedure TForm1.File1OnOpen(Sender: TObject);
var
  i: integer;
begin
  FWriteData := 'Hello World!' + #13#10;
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
end;

procedure TForm1.File1OnWrite(Sender: TObject);
var
  i: integer;
begin
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
  if Length(FWriteData) = 0 then begin
    File1.Close;
  end;
end;

procedure TForm1.File1OnClose(Sender: TObject);
begin
  Button1.Enabled := True;
end;
```

Take some time to try and understand what is going on. If you have used non-blocking sockets, you will understand this code pretty quickly. An outline follows:

1. Button1Click is called and opens the file. While the Open method returns immediately, the file is not open yet and cannot be accessed yet.
2. The OnOpen event is fired when the file has been opened and is ready to be accessed. An

attempt to write to the file is made, but all of the data may not be accepted. The Write method will return how many bytes were accepted. The rest of the data is saved and will be retried later.
3. The OnWrite event is fired when the file is ready for more data to be written and the Write method is attempted again with the remaining data.
4. Step 3 is repeated continually until all data has been accepted by the Write method. When all data has been accepted, the Close method is called. The file is not closed yet however.
5. The OnClose event is fired. The file is now closed.

5.17 File Write Comparison

Both examples only write data. Reading and writing data would further complicate the non-blocking example, but would only add one line of code to the blocking example.

For the non-blocking example, just to open, write data, and close the file it required:
- 3 File1 events
- 1 Form member variable

The non-blocking version is more complex and harder to understand. Given the choice between these two, very if any would
choose the non-blocking way. Most C++ programmers are excepted as many of them are simply masochists and would not choose either, because they are both too easy. Yet, nearly all socket components function using the non-blocking model.

5.18 Just Like Files

Using Indy is just like using Files. Actually Indy is often easier because Indy has dozens of methods for reading and writing instead of just a few. The Indy equivalent of the file example looks like this:

```
with IndyClient do begin
  Connect; Try
    WriteLn('Hello World.');
  finally Disconnect; end;
end;
```

As you can see, Indy really is very similar to working with files. The Connect replaces the Open, and the Disconnect replaces the Close. If you think and treat sockets like reading and writing to a file, you will find using Indy is quite easy.

Section VI

Introduction to Clients

6 Introduction to Clients

6.1 Basic Client

A basic Indy client takes this form:

```
with IndyClient do begin
  Host := 'test.atozedsoftware.com';
  Port := 6000;
  Connect; Try
    // Read and write data here
  finally Disconnect; end;
end;
```

The host and port can also be set at design time using the object inspector. This is the minimum code required to write a client using Indy. The minimum requirements for creating a client are as follows:

1. Set Host property.
2. Set Port property. This is only required if there is no default port. Most protocols provide a default port.
3. Connect.
4. Transfer Data. This includes reading and writing.
5. Disconnect.

6.2 Handling Exceptions

Handling exceptions with Indy clients is the same as it is with files. If an error occurs during a call to any Indy method, the appropriate exception will be raised. To handle these exceptions, code should be properly wrapped with try..finally or try..except blocks.

There are no OnError events, so do not go looking for them. This might seem strange if you have used other socket libraries, but consider TFileStream. TFileStream does not have an OnError event, it simply raises exceptions if there is a problem. Indy works in exactly the same manner.

Just as all file opens should be matched with a file close, all Connect calls in Indy should be matched with a call to Disconnect. Basic Indy clients should start out in this fashion:

```
Client.Connect; try
  // Perform read/write here
finally Client.Disconnect; end;
```

Indy exceptions are easy to differentiate from other VCL exceptions as all Indy exceptions descend from EIdException. If you wish to handle Indy errors separately from VCL errors, it can be done as in the following example.

Note: To use EIdException you will need to add IdException to your uses clause.

```
try
  Client.Connect; try
    // Perform read/write here
  finally Client.Disconnect; end;
except
  on E: EIdException do begin
    ShowMessage('Communication Exception: ' + E.Message);
  end else begin
    ShowMessage('VCL Exception: ' + E.Message);
  end;
```

```
end;
```

If there is an error during the call to the Connect method, it will clean itself up before throwing the appropriate exception. Because of this, the try is after the call to Connect and not before. However, if there is an exception during your data transfer only an exception will be raised. The socket will remain connected. It is your responsibility to retry your operation or disconnect. In the above form, no extra handling is performed and the socket is disconnected upon any error, and upon normal completion.

To also handle errors during the connect and separate them from other communication errors, your code should be of the form:

```
try
  IdTCPClient1.Connect; try
    try
      // Do your communications here
    finally IdTCPClient1.Disconnect; end;
  except
    on E: EIdException do begin
      ShowMessage('An network error occurred during communication: '
       + E.Message);
    end;
    on E: Exception do begin
      ShowMessage('An unknown error occurred during communication: '
       + E.Message);
    end;
  end;
except
  on E: EIdException do begin
    ShowMessage('An network error occurred while trying to connect: '
     + E.Message);
  end;
  on E: Exception do begin
    ShowMessage('An unknown error occurred while trying to connect: '
     + E.Message);
  end;
end;
```

This code not only checks for exceptions that occur during connection, but it separates such exceptions from exceptions that might occur during the communications. It further separates Indy exceptions from non-Indy exceptions.

6.3 Exceptions are not Errors

Many developers have been taught or have assumed that exceptions are errors. This is not the case. If this were the case Borland would have named them errors and not exceptions as they did.

An exception is merely something that is out of the ordinary. In terms of software, exceptions are something that occur and alter the normal program flow.

Exceptions are used to implement errors in Delphi, and thus most exceptions are errors. However there are exceptions such as EAbort which are not errors. Indy also defines several exceptions that are not errors. Such exceptions typically descend from EIdSilentException and thus can easily be distinguished from errors and other exceptions.

For an extended example please see EIdConnClosedGracefully.

6.4 TIdAntiFreeze

Indy has a special component that solves the user interface freeze problem transparently. The existence of a single TIdAntiFreeze instance in an application, allows the use of blocking calls in the

main thread without the User Interface being frozen.

The TIdAntiFreeze works by internally timing out calls to the stack and allowing messages to be processed during the timeout periods. The external calls to Indy continue to block, and thus code works exactly as without a TIdAntiFreeze.

Since the user interface freeze is only affected by blocking calls in the main thread, TIdAntiFreeze only affects Indy calls made from the main thread. If an application uses Indy in threads, TIdAntiFreeze is not required. If used, it will only affect calls made from the main thread.

Use of a TIdAntiFreeze will slow the socket communications somewhat. How much priority should be given to the application versus the socket communication is configurable with the properties of TIdAntiFreeze. The reason that usage of a TIdAntiFreeze slows the socket communication is that the main thread is allowed to process messages. Because of this care must be taken to not allow too much time to be consumed in events caused by messages. These include most user interface events such as OnClick, OnPaint, OnResize and many others. Since non-blocking sockets rely on messages themselves, the described problem always applies to non-blocking sockets. With Indy and the optional use of a TIdAntiFreeze, the programmer has complete control.

6.5 Demo - Postal Code Client

This demo is a client for a postal code lookup protocol. The protocol is very simple and for now the server is assumed to be predefined. The postal code lookup server is not covered in this section, only the client.

The purpose of the postal code lookup is to provide the city and state for a given postal code (Zip code for American readers). The sample data included with the server contains data for US postal codes. US postal codes (called zip codes) are 5 digits and numeric.

The server for this demo is covered later.

6.5.1 Postal Code Protocol

The postal code protocol is very simple. It consists of only two commands:

- Lookup <post code 1> <post code 2> ...
- Quit

A sample conversation looks like this:

```
Server: 204 Post Code Server Ready.
Client: lookup 16412
Server: 200 Ok
Server: 16412: EDINBORO, PA
Server: .
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
Client: quit
Server: 201-Paka!
Server: 201 4 requests processed.
```

The server responds with a greeting when the client connects. Greetings and replies to commands typically contain a 3 digit number specifying status. This will be covered more in detail in later

sections.

After the greeting the server is ready to accept commands from the client. If a Lookup command is received the server will respond with a list of post codes followed by the associated location. The response is terminated by a . on a line by itself. The client can issue as many commands as it likes until the client specifies it is ready to disconnect by issuing the Quit comand.

6.5.2 Code Explanation

The postal code client consists of two buttons, a listbox, and a memo. One button is used to clear results, and the other is used to take values from the memo and request information from the server. The results are displayed in the listbox.

In a normal application the user would have options to specify the host, port and possibly even proxy configuration information. However for the purposes of demonstration the host has been set at design time to 127.0.0.1 (Self) and the port to 6000.

When the user clicks on the Lookup button the following event will be executed:

```
procedure TformMain.butnLookupClick(Sender: TObject);
var
  i: integer;
begin
  butnLookup.Enabled := true; try
    lboxResults.Clear;
    with Client do begin
      Connect; try
        // Read the welcome message
        GetResponse(204);
        lboxResults.Items.AddStrings(LastCmdResult.Text);
        lboxResults.Items.Add('');
        // Submit each zip code and read the result
        for i := 0 to memoInput.Lines.Count - 1 do begin
          SendCmd('Lookup ' + memoInput.Lines[i], 200);
          Capture(lboxResults.Items);
          lboxResults.Items.Add('');
        end;
        SendCmd('Quit', 201);
      finally Disconnect; end;
    end;
  finally butnLookup.Enabled := True; end;
end;
```

The Indy methods used here will be explained only briefly as they are covered in more detail in other sections.

The code starts by disabling the button so that the user cannot try to perform another lookup while one is in progress. You might think that this is not possible because button click events are processed using messages. However this demo has a TIdAntiFreeze which calls Application.ProcessMessages and allows paint events as well as other click events to occur. Because of this you need to take extra care to protect the state of the user's actions.

Using a TIdTCPClient (Client) dropped on the form at design time the demo then connects to the server and waits for the greeting. GetResponse reads the response and returns the response as the result. In this case the result is discarded, but GetResult is instructed to verify that the numeric response is 204. If the server responds with a different code an exception will be raised. A server may respond with a different code if it is too busy, under maintenance, etc.

For each postal code that the user enters, the demo sends a lookup command to the server and expects a 200 reply, followed by a response. SendCmd sends the command, then calls GetResponse

passing the second parameter (in this case 200). If SendCmd succeeds, the demo calls Capture which reads a response until a . is found on a line by itself. Since the demo is only submitting one postal code per command, it expects only one line for a reply, or none if the postal code is invalid.

Finally the demo sends the Quit command, waits for a 201 reply which acknowledges that the server agrees and disconnects. It is always a good idea to use a Quit command so that both sides know a disconnect is about to occur.

Section VII

UDP

7 UDP

7.1 Overview

UDP (User Datagram Protocol) is for datagrams and is connectionless. UDP allows lightweight packets to be sent to a host without having to first connect to another host. UDP packets are not guaranteed to arrive at their destination, and may not arrive in the same order they were sent. When sending a UDP packet, it is sent in one block. Therefore, you must not exceed the maximum packet size specified by your TCP/IP stack.

Because of these factors, many people assume UDP is nearly useless. This is not the case. Many streaming protocols, such as Real Audio, use UDP.

Note: The term "streaming" can be easily confused with "stream" connection, which is TCP. When you see these terms, you need to determine the context in which they are used to determine their proper meaning.

7.2 Reliability

The reliability of UDP packets depends on the reliability and the saturation of the network. UDP packets are often used on applications that run on a LAN, as the LAN is very reliable. UDP packets across the Internet are generally reliable as well and can be used with error correction or more often interpolation. Interpolation is when an educated guess is made about missing data based on packets received before and / or after. Delivery however cannot be guaranteed on any network - so do not assume your data will always arrive at your destination.

Because UDP does not have delivery confirmation, its not guaranteed to arrive. If you send a UDP packet to another host, you have no way of knowing if it actually arrived at its destination. The stack will not - and cannot - determine this, and thus will not provide an error if the packet did not reach its destination. If you need this information, you need to send some sort of return notification back from the remote host.

UDP is like sending someone a message on a traditional pager. You know you sent it, but you do not know if they received it. The pager may not exist, may be out of the service area, may not be on, or may not be functioning. In addition, the pager network may lose the page. Unless the person pages you back, you do not know if your message was delivered. In addition, if you send multiple pages, it is possible for them to arrive out of order.

Another real world example that is similar to UDP, is the postal service. You can send it, but you cannot guarantee it will be delivered to the destination. It may be lost anywhere along the way, or delivered, but mutilated before delivery.

7.3 Broadcasts

UDP has a unique ability that is often the feature that makes it desirable. This ability is the ability to be broadcasted. Broadcasting means that a single message can be sent, but can be received by many recipients. This is not the same as multicasting. Multicasting is a subscription model where recipients subscribe and are added to a distribution list. With broadcasting, a message is sent across the network and anyone listening can receive without the need to subscribe.

Multicasting is similar to a newspaper delivery. Only people who subscribe to the newspaper receive it. Broadcasting is similar to a radio signal. Anyone with a receiver can tune to a specific radio station

and receive it. The user does not need to notify the radio station that they wish to listen.

A specific broadcast IP can be calculated based on the IP of the sender and a subnet mask. However in most cases it is easiest to use 255.255.255.255 which will broadcast as far as possible.

Nearly all routers however are programmed to filter out broadcast messages by default. This means that messages will not pass across bridges or external routes, and the broadcast will be limited to the local LAN network.

7.4 Packet Sizes

Most operating systems allow UDP packet sizes of 32K or even 64K. However typically routers will have smaller limits. UDP packets can only be as big as the maximum allowable size that is permitted by any router or network device along the route that the UDP packet must travel. There is no way to know this value or predict it.

Because of this, it is recommended that UDP packets be kept at 8192 bytes or less if you are transmitting beyond the local LAN. In many cases even this may be too large of a value. To be absolutely sure, keep all UDP packets 1024 bytes or less.

7.5 Confirmations

7.5.1 Overview

In a LAN environment UDP is quite reliable. However when WAN's or the Internet is involved you may wish to implement a variety of confirmation schemes.

7.5.2 Acknowledgements

In the acknowledgement system, each packet is acknowledged by the receiver as having been received. If no acknowledgement is received within a given time period after transmission, it is retransmitted.

Because the acknowledgements themselves may be lost, each packet should have a unique identifier. Normally this identifier is simply a sequence number. This is so that the recipient can filter duplicate packets if it receives them, and also so the acknowledgement messages can identify the packet that they are acknowledging receipt of.

7.5.3 Sequencing

Packets can be identified with a sequence number. This number can be used by the recipient to determine if packets are missing. It can then re-request specific missing packets be resent, or in some cases such as audio it can interpolate data from surrounding packets and make a best guess at the missing packet, or simply ignore the lost packet.

This behavior of missing packets can be heard with real audio, or a digital mobile (cell) phone when packets are lost. In some systems you merely hear skipping or gaps. In other cases where the packets are small and the system interpolates, you may hear a "worbling" sound.

7.6 TIdUDPClient

TIdUDPClient is the base UDP client for sending UDP packets to other destinations. The most commonly used method is Send, which uses the Host and Port properties to send a UDP packet. It accepts a string as an argument.

There is also a SendBuffer method which performs the same task as Send, except that it accepts a Buffer and Size as arguments.

TIdUDPClient can also be used as a server of sorts to wait and receive incoming UDP packets on an individual basis.

7.7 TIdUDPServer

Since UDP is connectionless, TIdUDPServer operates differently than TIdTCPServer. TIdUDPServer does not have any modes similar to TIdSimpleServer, but since UDP is connectionless, TIdUDPClient does have single use listening methods.

TIdUDPServer when active creates a listening thread to listen for inbound UDP packets. For each UDP packet received, TIdUDPServer will fire the OnUDPRead event in the main thread, or in the context of the listening thread depending on the value of the ThreadedEvent property.

When ThreadedEvent is false, the OnUDPRead event will be fired in the context of the main program thread. When ThreadedEvent is true, the OnUDPRead event is fired in the context of the listener thread.

When ThreadedEvent is true or false, its execution will block the receiving of more messages. Because of this the processing of the OnUDPRead event must be quick.

7.8 UDP Example - RBSOD

7.8.1 Overview

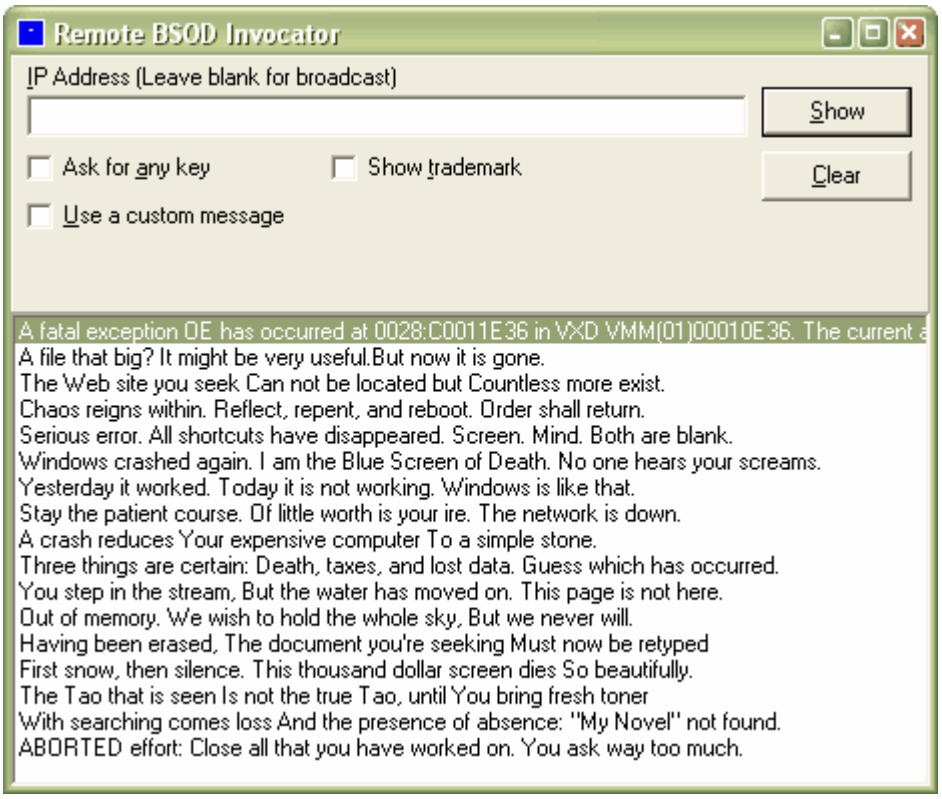
This example will provide an example of a UDP client and a UDP server. The example is a useful application as well which can be used to spread fun around many corporate environments. However be sure to use it with care.

The example is named Remote BSOD invocator - or RBSOD for short. RBSOD can be used to trigger fake BSODs on colleagues (or enemies) machines. The BSOD's that are triggered are not real, and will not cause the person to lose any data. However they should give them a startle. The BSOD's can also be cleared remotely.

RBSOD consists of two programs. The server which is to be run on the remote machine, and the client which is used to control and trigger the BSOD's.

Many options are provided. Depending on the options that are selected, the BSOD can be made to appear quite authentic, or it can be made to appear as a joke. However when they are made to appear as a joke it will take the user a few seconds to understand this and provide them with an initial startle.

The client (RBSOD) looks like this:



The following options are provided.

IP Address

Specifies the IP address or host name of the computer you that wish to trigger the BSOD. The server must be installed and running on the remote computer.

If this field is left empty, the message will be broadcast across the local subnet (in most cases your local LAN) and all computers which have the server installed and running will display the BSOD.

Message

Message is the error message that will be displayed to the user in the BSOD screen. The default message is:

```
A fatal exception OE has occurred at 0028:C0011E36 in VXD VMM(01)00010E36. The
current application will be terminated.
```

This text was taken from a real BSOD, and will appear quite real to the user.

Many Japanese Haiku's (A special form of a Japanese poem) are also included for fun. There are many very funny ones to choose from. Here two of my favorites:

```
Windows crashed again. I am the Blue Screen of Death. No one hears your screams.

Three things are certain: Death, taxes, and lost data. Guess which has occurred.
```

Imagine the look on your colleagues faces after they realize they have been fooled. Be very careful when you perform this prank on Dilbert type bosses or Visual Basic programmers however. They may

not realize that it is a prank.

These messages are contained in messages.dat, and you can add your own as well.

Use a Custom Message

If this option is checked, another text box will appear and you can enter a custom message. This option is useful for providing interactive or relevant BSOD messages. For example if your boss is wearing an unusually tacky suit one day you might trigger this message:

```
Ugly brown suit error. I cannot continue in such company.
```

Show Any Key

This option can be used to additional humor to the prank. The BSOD will initially prompt the user with "Press any key to continue _". This is the same as a normal BSOD. However if this option is checked, after they press a key it will then further prompt them with flashing text "Not that key, press the NY key!".

This option is sure to keep Visual Basic programmers and Dilbert bosses busy for hours hunting for the Any key. This option is best used just prior to leaving for the airport on a long business trip, or when you need to keep them busy for a while.

Show Trademark

If this option is used, in the bottom corner of the screen the following the following text will be displayed in small but readable text:

```
* The BSOD is a trademark of the Microsoft Corporation.
```

Show

The show button will trigger the BSOD on the remote computers.

Clear

Clear can be used to remotely clear the triggered BSODs. Normally you will let the end user clear their own BSOD, but in some cases you may wish to clear it remotely.

7.8.2 Server

Let's take a look at the server first.

Installation

The server is named svchost instead of RBSODServer or some other sensible name. This is so that you can install it easily on other computers while hiding it. svchost is a normal windows executable of which normally multiple copies are executing. If you look in task manager now, you are likely to see four entries for this executable. Since you will put this special version in its own directory, it will not interfere with the windows one, but will appear the same as the normal when viewed in task manager.

The server does not have any windows and will not appear in the system tray or task tray. If you want to stop it, use the task manager and select the svchost that is running as the logged on user. The system svchosts will be running as SYSTEM. When you have selected your version, you can use END TASK.

For most installations simply copy the compiled version of the server to the target computer (compile without packages for easiest deployment) and run it. It will stay in memory until the user restarts their computer. After they restart, the program will not reload. If you want the program to automatically reload you can add it to the users start up group or use registry entries for a stealthier start.

Source Code

The server consists of two units, Server.pas and BSOD.pas. BSOD.pas contains a form that is used to display the BSOD screen. BSOD.pas does not contain any Indy code and thus will not be covered here.

Server.pas is the main application form and contains a single UDP server. The port property has been set to 6001 and the active property has been set to True. When the application runs, it will immediately begin listening on port 6001 for incoming UDP packets.

As discussed previously UDP is similar to a pager. Because of this there are no connections required to receive data. Data packets simply arrive as one piece. For each UDP packet that is received, the UDP server will fire the OnUDPRead event. No other events are needed to implement a UDP server. When the OnUDPRead event is fired, the complete packet will have been received and is now available for use.

The OnUDPRead event passes in three arguments:

1. ASender: TObject - This is the component which fired this event. This is only useful multiple UDPServers have been created and are sharing a single method as an event. This is rarely used or needed.
2. AData: TStream - This is the main argument and it contains the packet. UDP packets can contain text and/or binary data. Because of this Indy represents them as a stream. To access the data, simply use the read methods of the TStream class.
3. ABinding: TIdSocketHandle - This is useful for advanced options to retrieve information about the binding that was used to receive the packet. This is only useful if you have created bindings.

Here is what the OnUDPRead event looks like for the RBSOD server:

```
procedure TformMain.IdUDPServer1UDPRead(Sender: TObject; AData: TStream;
  ABinding: TIdSocketHandle);
var
  LMsg: string;
begin
  if AData.Size = 0 then begin
    formBSOD.Hide;
  end else begin
    // Move from stream into a string
    SetLength(LMsg, AData.Size);
    AData.ReadBuffer(LMsg[1], Length(LMsg));
    //
    formBSOD.ShowBSOD(Copy(LMsg, 3, MaxInt)
     , Copy(LMsg, 1, 1) = 'T'
     , Copy(LMsg, 2, 1) = 'T');
  end;
end;
```

Notice that there is an if statement that checks to see if the size is 0. It is legal to send and receive empty UDP packets. In this case it is used to signal the server to clear the BSOD.

If the size is not 0, the data is read into a local string using TStream.ReadBuffer.

The UDP server does not use an individual thread for each packet, so OnUDPRead events occur sequentially. By default OnUDPRead is fired in the main thread so forms and other GUI controls can be accessed safely.

7.8.3 Client

The RBSOD client is even simpler than the server. The RBSOD client consists of one form: Main.pas. Main.pas contains several events but most of them are related to the user interface and should be

self explanatory.

The relevant Indy code in the RBSOD client is this excerpt which is taken from the OnClick of the Show button.

```
IdUDPClient1.Host := editHost.Text;
IdUDPClient1.Send(
 iif(chckShowAnyKey.Checked, 'T', 'F')
 + iif(chckTrademark.Checked, 'T', 'F')
 + s);
```

The first line sets the host that the UDP packet will be sent to. The port has already been set at design time using the property inspector and matches that of the server with the value of 6001.

The next line uses the Send method to send the UDP packet. Since UDP is connectionless, any data must either be sent as multiple packets, or packaged into a single packet. If multiple packets are sent, it is the developers responsibility to coordinate and reassemble them. This is not as trivial a task as it appears, so unless you are sending large amounts of data it is much easier to assemble the data into a single packet.

The argument passed to send will be immediately sent as a UDP packet and thus when it is called, all data that you want to be sent must be passed in a single call.

Indy also contains an overloaded SendBuffer method to send data using buffers.

In the case of RBSOD the protocol simply consists of two characters that specify options to show the trademark, and show any key, followed by the actual message to display.

The only other Indy code in the RBSOD client is in the OnClick for the Clear button and it is nearly identical to the previous excerpt.

Section VIII

Reading and Writing

8 Reading and Writing

Indy supports many methods for reading and writing data to meet many different needs. These methods include methods to wait, check status, and poll.

Each of these methods belongs to the TIdTCPConnection class. This is the type of each server connection, and an ancestor of TIdTCPClient. This means that you can use all of these methods in both servers and in clients.

Most people are familiar with only a few of the read and write methods. Furthermore, many people never use the core clients, but only protocol clients and thus may not be familiar with any of the core methods.

Remember, Indy is blocking so do not look for events to notify you when the requested operation is complete. Methods do not return until their requested task is complete. If the requested task cannot be completed, an exception will occur.

This is not a full set of documentation for each method. For that you should refer to the Indy Help file. This section is merely designed to familiarize you with the methods.

8.1 Read Methods

8.1.1 AllData

```
function AllData: string;
```

AllData blocks and collects incoming data until the connection has been disconnected. It then returns the collected data as the result. AllData is useful for protocols such as WhoIs which after given input return data and signal the end of data by disconnecting. AllData buffers data in memory so it should not be used with large amounts of data.

8.1.2 Capture

```
procedure Capture(ADest: TStream; const ADelim: string = '.';
const AIsRFCMessage: Boolean = True); overload;

procedure Capture(ADest: TStream; out VLineCount: Integer; const
ADelim: string = '.'; const AIsRFCMessage: Boolean = True);
overload;

procedure Capture(ADest: TStrings; const ADelim: string = '.';
const AIsRFCMessage: Boolean = True); overload;

procedure Capture(ADest: TStrings; out VLineCount: Integer; const
ADelim: string = '.'; const AIsRFCMessage: Boolean = True);
overload;
```

Capture has several overloaded forms. In summary, Capture will read data until the specified delimiter is found on a line by itself.

8.1.3 CurrentReadBuffer

```
function CurrentReadBuffer: string;
```

CurrentReadBuffer returns all data currently contained in Indy's internal receive buffer. Before returning data, CurrentReadBuffer will also attempt to read any and all data that is also waiting from the socket if connected. Calling CurrentReadBuffer empties the internal buffer.

If no data is available, an empty string will be returned.

8.1.4 InputBuffer

```
property InputBuffer: TIdManagedBuffer read FInputBuffer;
```

InputBuffer is a reference to an instance of a TIdManagedBuffer object. InputBuffer is Indy's internal receive buffer. TIdManagedBuffer does have several advanced methods, but typically, users do not need to call these.

8.1.5 InputLn

```
function InputLn(const AMask: String = ''; AEcho: Boolean = True; ATabWidth: Integer
= 8; AMaxLineLength: Integer = -1): String;
```

InputLn reads a line from the server and echoes it back honoring the backspace character. If AMask is specified, the AMask string is sent to the peer for each character received, instead of the actual characters that are read. InputLn is useful when obtaining data that you do not want displayed to the

user, such as during password entry.

8.1.6 ReadBuffer

```
procedure ReadBuffer(var ABuffer; const AByteCount: Longint);
```

ReadBuffer is a used to read data directly into a specified memory buffer. ReadBuffer will read data from the socket if not enough data is available in Indy's internal buffer.

8.1.7 ReadCardinal

```
function ReadCardinal(const AConvert: boolean = true): Cardinal;
```

ReadCardinal reads an unsigned 32-bit cardinal from the connection, optionally adjusting for network byte order.

8.1.8 ReadFromStack

```
function ReadFromStack(const ARaiseExceptionIfDisconnected: Boolean = True;
ATimeout: Integer = IdTimeoutDefault;
```

ReadFromStack is used to fill Indy's internal receive buffer. Normally end users should never call this function unless they are implementing a new read method that does not use other existing read methods, or they are directly interacting with the internal receive buffer by using the InternalBuffer property.

8.1.9 ReadInteger

```
function ReadInteger(const AConvert: boolean = true): Integer;
```

ReadInteger reads a signed 32-bit integer from the connection, optionally adjusting for network byte order.

8.1.10 ReadLn

```
function ReadLn(ATerminator: string = LF; const ATimeout: Integer =
IdTimeoutDefault; AMaxLineLength: Integer = -1): string; virtual;
```

ReadLn reads data from the connection until the terminator is found, the timeout period elapses, or the maximum line length is reached.

8.1.11 ReadLnWait

```
function ReadLnWait(AFailCount: Integer = MaxInt): string;
```

ReadLnWait functions like ReadLn with the exception that it will not return until a non\-empty line is found. It will also return if the AFailCount empty lines are read.

8.1.12 ReadSmallInt

```
function ReadSmallInt(const AConvert: Boolean = true): SmallInt;
```

ReadSmallInt reads a small integer from the connection, optionally adjusting for network byte order.

8.1.13 ReadStream

```
procedure ReadStream(AStream: TStream; AByteCount: LongInt = -1; const
AReadUntilDisconnect: boolean = false);
```

ReadStream reads data into a stream. A specific byte count can be specified, read from the stream, or the stream can be read until disconnect.

8.1.14 ReadString

```
function ReadString(const ABytes: Integer): string;
```

ReadString reads the specified number of bytes into a string and returns the data as the result.

8.1.15 ReadStrings

```
procedure ReadStrings(var AValue: TStrings; AReadLinesCount: Integer = -1);
```

ReadStrings reads a specified number of strings terminated by EOLs from the connection. If a string count is not specified, a 32 bit integer is first read from the connection and that number is used.

8.1.16 WaitFor

```
function WaitFor(const AString: string): string;
```

WaitFor reads data from the connection until the specified string is found.

8.2 Read Timeouts

TIdTCPConnection (All TCP clients and the connections that servers use descend from TIdTCPConnection) has a property named ReadTimeout. ReadTimeout specifies the desired timeout in milliseconds. The default property value is IdTimeoutInfinite. This setting disables read timeout functionality.

The timeout is not a timeout for a complete unit of work. It is an idle timeout. That is, if the amount of time specified in the ReadTimeout property passes and no piece of the data can be written, an EIdReadTimeout exception will be raised.

Many times a network connection becomes very slow or transfers no data, but remains connected and valid. In such situations, the connection may become so slow as to become unusable and only becomes a burden for the server and useless to the client.

To deal with these situations Indy implements read timeouts by using the ReadTimeout property of TIdTCPConnection. ReadTimeout defaults to 0 which disables read timeouts. To enable a read timeout specify a value in milliseconds.

During any read call, if the specified read timeout and no data is received from the connection, an EIdReadTimeout exception will be raised. The timeout does not apply to the data requested being received with in the timeout period, but any data. If you request 100 bytes of data to be read and a read timeout of 1000 milliseconds (1 second) the read operation may still take longer than 1 second. Only if 1 second passes without a single byte being received will the EIdReadTimeout exception will be raised.

8.3 Write Methods

8.3.1 SendCmd

```
function SendCmd(const AOut: string; const AResponse: SmallInt = -1): SmallInt;
overload;

function SendCmd(const AOut: string; const AResponse: Array of SmallInt): SmallInt;
overload;
```

SendCmd is used to send a text command and parse an RFC style numeric reply.

8.3.2 Write

```
procedure Write(AOut: string);
```

Write is the most basic output method in Indy. Write sends the AOut parameter to the connection. Write does not alter the AOut parameter in any way.

8.3.3 WriteBuffer

```
procedure WriteBuffer(const ABuffer; AByteCount: Longint; const AWriteNow: Boolean =
False);
```

WriteBuffer allows a direct memory buffer to be written. If AWriteNow is specified, write buffering will be bypassed if it is currently in use.

8.3.4 WriteCardinal

```
procedure WriteCardinal(AValue: Cardinal; const AConvert: Boolean = True);
```

WriteCardinal writes an unsigned 32-bit cardinal to the connection, optionally adjusting for network byte order.

8.3.5 WriteHeader

```
procedure WriteHeader(AHeader: TStrings);
```

WriteHeader writes a TStrings object to the connection converting the '=' to a ': ' sequence for each item. WriteHeader also writes a blank line after writing out the TStrings object.

8.3.6 WriteInteger

```
procedure WriteInteger(AValue: Integer; const AConvert: Boolean = True);
```

WriteInteger writes a signed 32-bit integer to the connection, optionally adjusting for network byte order.

8.3.7 WriteLn

```
procedure WriteLn(const AOut: string = '');
```

WriteLn performs the same function as Write, with the exception that it also writes an EOL (CR + LF) after the AOut parameter.

8.3.8 WriteRFCReply

```
procedure WriteRFCReply(AReply: TIdRFCReply);
```

WriteRFCReply writes out a numeric + text RFC style reply using the specified TIdRFCReply object.

8.3.9 WriteRFCStrings

```
procedure WriteRFCStrings(AStrings: TStrings);
```

WriteRFCStrings writes out the specified TStrings object in RFC message format terminating the strings with a '.' on a line by itself.

8.3.10 WriteSmallInt

```
procedure WriteSmallInt(AValue: SmallInt; const AConvert: Boolean = True);
```

WriteSmallInt writes a small integer to the connection, optionally adjusting for network byte order.

8.3.11 WriteStream

```
procedure WriteStream(AStream: TStream; const AAll: Boolean = True; const
AWriteByteCount: Boolean = False; const ASize: Integer = 0);
```

WriteStream writes a specified stream to the connection. WriteStream contains many parameters for specifying what parts of the stream are written, and can optionally write the byte count to the connection as well.

8.3.12 WriteStrings

```
procedure WriteStrings(AValue: TStrings; const AWriteLinesCount: Boolean = False);
```

WriteStrings writes a TStrings to the connection and is the counterpart to ReadStrings.

8.3.13 WriteFile

```
function WriteFile(AFile: String; const AEnableTransferFile: Boolean = False):
Cardinal;
```

WriteFile is a function to write the contents of a file directly to the connection. WriteFile uses operating system optimizations to write the file to the socket that produces better performance than

simply using a TFileStream with SendStream.

8.4 Write Buffering

TCP must send data in packets. The size of the packets can vary but is usually a little more than 1 kilobyte. However, if TCP waited for a full packet of data, in many cases responses could not be returned because data would not be sent. So that TCP can use packets as required and not require a full packet before sending, an algorithm called Nagle Coalescing is used. Nagle Coalescing buffers data internally until the packet size has been reached, or an internally calculated amount of time has passed. The time period is usually quite small and in milliseconds.

Sending many small pieces of data can confuse the algorithm or cause it to act cause too many packets to be sent. Since each packet also has overhead, this wastes bandwidth and slows down the data transfer.

Data could be buffered in a string or other and sent at once, however this would put the requirement on your code and also quite likely alter the way you sent data as you would not have all the write methods available during buffering. It would also complicate your code and increase memory usage.

Instead, you can use Indy's write buffering feature. By using write buffering, you are allowing Indy to buffer outgoing data and allow you to use Indy normally with all of its write calls enabled.

To begin write buffering, call OpenWriteBuffer and specify a buffer size. Then you may call any of Indy's write functions, and all output is buffered until the buffer size is reached. Each time the buffer size is reached the buffer will be written to the connection and cleared. If no buffer size is specified, all data will be buffered until manually flushed.

There are also several write buffering methods for managing the buffer.

ClearWriteBuffer clears the current buffer and keeps the buffer open. FlushWriteBuffer flushes the current contents and also keeps the buffer open.

To finish write buffering, call CancelWriteBuffer or CloseWriteBuffer. CloseWriteBuffer writes any remaining data and concludes write buffering, while CancelWriteBuffer closes the buffer without transmitting what it contains.

8.5 Work Transactions

Work transactions are used to define units of work. They are called work transactions as they may be nested, and seperate read and write transactions can occur simultaneously. Work transactions are commonly used to display progress bars or transfer status displays.

Transactions are predefined by various methods in Indy such as TIdHTTP.Get, WriteStream, etc. As a user, you can also define your own transactions using BeginWork, DoWork and EndWork .

8.5.1 OnWork Events

The OnWork events consist of three events and are used to communicate the status of a work transaction. These three events are: OnWorkBegin, OnWork and OnWorkEnd.

When a transaction begins, an OnWorkBegin event will be fired. The OnWorkBegin event specifies

the Sender, WorkMode, and WorkCount. The Sender is the connection to which the transaction applies. The WorkMode specifies if it is a read transaction or a write transaction. Read and Write transactions can occur concurrently, and transactions can be nested. The WorkCount specifies the size of the transaction. In many transactions, the size cannot be known beforehand and in such cases, WorkCount has a value of 0. Otherwise, WorkCount specifies the number of bytes. Usually this event is used for preparing progress bars.

Next, a series of OnWork events is fired. The OnWork event specifies the Sender, WorkMode, and the current WorkCount. This is event is useful for updating progress bars.

When the transaction is complete, the OnWorkEnd event is fired. The OnWorkEnd even specifies only the Sender and the WorkMode. This event is useful for marking progress bars complete.

8.5.2 Managing Your Own Work Transactions

You can create your own transactions as well by calling BeginWork, DoWork and EndWork. The parameters are the same as the events. The Work calls manage the nesting automatically if you nest transactions.

To perform a transaction, first call BeginWork with the size of the transaction if known. Then call DoWork with updates on progress. When finished call EndWork.

Section

Detecting Disconnects

9 Detecting Disconnects

Because Indy is blocking in nature and its events are status related only, there is no event to notify of a prematurely disconnected connection. If a read or write call is in progress when a premature disconnect occurs, an exception will be raised and can be caught. If no read or write call is in progress, no exception will be raised until a read or write call is attempted.

There is an OnDisconnected event, however this is not what you might think it is. OnDisconnected is called when the Disconnect method is called. It is not an event telling you of a premature disconnection.

9.1 Saying Good Bye

In TCP command based protocols the easiest way is to say good bye. This will only detect when a proper connection has occurred, but this is a big help.

To say good bye, protocols use the QUIT command. When a client is ready to disconnect instead of merely just disconnecting, it first issues a QUIT command the server. It then waits for a response and then disconnects the socket itself.

Its good manners and allows both sides to properly disconnect. Not doing so is like talking on a phone, and unexpectedly the other party hangs up. You do not know what happened and are left guessing until you decide to hang up too.

The difference is that a server may have hundreds or thousands of clients. If they all disconnect without telling the server, the server will have a lot of "dead" connections to reconcile.

9.2 Do you really need to know?

Many programmers instantly balk at this fact, contending that they need to know immediately when a connection has been disconnected prematurely. You may have heard the saying, "If a tree falls in the forest and no one is present to hear it, does it make a sound?" So, if a socket disconnects, and it is not being accessed, does it really matter if it is closed? In most cases, the answer is no. If a socket is prematurely disconnected and it is not being accessed, no exception will be raised and no event will be fired until the socket is accessed.

This may seem strange, but this is the same as if we were accessing a file. Imagine that you have an excel spreadsheet open that exists on a floppy disk. You are working on the file, but it is cached in memory and excel is not reading or writing at the current moment. Sometime during this time, you forget that you have the spreadsheet open and you remove the floppy disk from the drive. Later on, you return to working on your spreadsheet. All is fine until you go to save your changes. Only then do you receive an error. Excel did not receive an "EAnIdiotRemovedTheFloppyDiskException" when you removed the floppy disk, even though Excel had a file open on that disk.

9.3 I need to know now!

Note that the exception may not occur immediately. For instance, if you pull the network cable the exception will not occur immediately, but may take a minute or more. TCP/IP was designed by the US Military to be a redundant network protocol to withstand nuclear attacks. Because of its design, the network will wait for time outs from the other side of the connection, as well as attempting retries. For

you to detect disconnections immediately is contrary to the way that TCP/IP was designed. Such detection can be implemented, but you it is important that you understand why it does not function this way by default. Your understanding of this will help you to properly implement immediate disconnection detect

In most cases if you think about it, you will find that this behaviour is acceptable and even desirable. However there are situations that it is very important to know if a connection has been lost. Imagine that you are implementing the interface between a heart monitor and a telemetry monitoring system. This is one case that you definitely want to know if the connection has been lost, and very quickly.

If you need immediate notification of disconnects you will need to implement such detection into the protocol.

9.3.1 Keep Alives

Keep alives are one method to detect immediate disconnects. Keep alives are implemented such that one side of a connection on a regular interval sends a message such as NOOP (No Operation) to the other side and awaits a response. If a response is not received in a specified amount of time, the connection is considered problematic and terminated. The timeout period is normally quite short and specific to the needs of the protocol it is used in. If you are implementing the connection to a heart monitor, it is hoped that the specified timeout period would be shorter than if you were implementing a beer keg temperature monitor.

There is and additional advantage gained by implementing a keep-alive system. In many cases, a TCP connection may remain valid however one of the processes stops responding and stops servicing requests. In other cases, the process may continue servicing requests but at a rate too slow to be effective either because of problems in the service, or degrading network bandwidth. Keep-alives will detect this condition and mark it as unsuitable for use, even if the connection is still valid.

Keep-alives may be sent on regular intervals, or only at times when it is necessary to determine the status depending on the needs of the protocol.

9.3.2 Pings

Pings are implemented similar to keep alives with the exception that they are not returned in response to a request. Pings are sent from one side of the connection to the other on regular intervals. One side becomes the broadcaster and one side becomes the monitor. If a specified amount of time elapses on the monitor since the last ping arrived without a new one arriving, the connection is determined to be unsuitable and terminated.

Pings can be thought of as a pulse or a heartbeat. If it stops or slows, there is a problem.

9.4 EIdConnClosedGracefully

Many Indy users are annoyed by the EIdConnClosedGracefully exception that is raised with Indy servers, especially the HTTP and other servers. EIdConnClosedGracefully is an exception signaling that the connection has been closed by the other side intentionally. This is not the same as a broken connection which would cause a connection reset error. If the other side has closed the connection and the socket is read or written to, EIdConnClosedGracefully will be raised by Indy. This is similar to attempting to read or write to a file that has been closed without your knowledge.

In some cases this is a true exception and your code needs to handle it. In other cases (typically servers) this is a normal part of the functioning of the protocol and Indy handles this exception for you. Even though Indy catches it, when running in the IDE the debugger will be triggered first. You

can simply press F9 to continue and Indy will handle the exception, but the constant stopping during debugging can be quite annoying. In the cases where Indy catches the exception, your users will never see an exception in your program unless it is run from the IDE.

9.4.1 Introduction

9.4.2 Why Does This Exception Occur in Servers?

When a client is connected to a server there are two common ways to handle the disconnection:

1. Mutual Agreement - Both sides agree to mutually disconnect by one side signaling (and the other optionally acknowledging) and then both sides disconnecting explicitly.
2. Single Disconnect - Disconnect and let the remote side take notice.

With the Mutual Agreement method both sides know when to disconnect and both explicitly disconnect. Most conversational protocols such as Mail, News, etc disconnect in this manner. When the client is ready to disconnect it sends a command to the server telling it that it will disconnect. The server replies with an acknowledgement of the disconnect request, and then both the client and server disconnect. In these cases an EIdConnClosedGracefully should not be raised, and if one occurs it is in fact an error and should be handled.

In some cases of Mutual Disconnect no command will be issued, but both sides know when the other will disconnect. Often a client will connect, issue one command, receive the response from the server and disconnect. While no explicit command was issued by the client, the protocol states that the connection should be disconnected after one command and response. Some of the time protocols are examples of this.

With Single Disconnect, one side just disconnects. The other side is left to detect this and then take appropriate action to terminate the session. With protocols that use this disconnection method you will see EIdConnClosedGracefully and it is normal. It is an exception, but Indy knows about it and will handle it for you. The whois protocol is an example of this. The client connects to the server and sends a string containing the domain to query. The server then sends the response and disconnects when the response is finished. No other signal is sent to the client other than a normal disconnection that the response is finished.

The HTTP allows for both Mutual Agreement and Single Disconnect and this is why it is common to see the EIdConnClosedGracefully with the HTTP server. HTTP 1.0 works similar to the whois protocol in that the server signals the client simply by disconnecting after the request has been served. The client then must use new connections for each request.

HTTP 1.1 allows a single connection to request multiple documents. However there is no command to signal a disconnect. At any time, either the client or the server can disconnect after a response. When the client disconnects but the server is still accepting requests, a EIdConnClosedGracefully will be raised. In most cases in HTTP 1.1, the client is the one that disconnects. A server will disconnect when it implements part of HTTP 1.1, but does not support the keep alive option.

9.4.3 Why is it an Exception?

Many users have commented that maybe it there should be a return value to signal this condition instead of an exception. However this is the wrong approach in this case.

The EIdConnClosedGracefully is raised from a core routine, however when this routine is called it is normally several method calls deep. The EIdConnClosedGracefully is in fact an exception and needs to be trapped by the topmost caller in most cases. The proper way to handle this is an exception.

9.4.4 Is it an Error?

All exceptions are not errors. Many developers have been taught or assumed that all exceptions are errors. However this is not the case, and this is why they are called exceptions and not errors.

Exceptions are exactly that - exceptions. Delphi and C++ Builder use exceptions to handle errors in an elegant way. However exceptions have other uses besides errors as well. EAbort is one example of an exception that is not necessarily an error. Exceptions such as these are used to modify standard program flow and communicate information to a higher calling level where they are trapped. Indy uses exceptions in such a way as well.

9.4.5 When is it an Error?

When EIdConnClosedGracefully is raised in a client, it is an error and you should trap and handle this exception.

In servers it is an exception. However sometimes it is an error, and sometimes it is an exception. For many protocols this exception is part of the normal functioning of the protocol. Because of this common behavior, if you do not catch the EIdConnClosedGracefully in your server code, Indy will. It will then mark the connection as closed and stop the thread assigned to the connection. If you wish to handle this exception yourself you may, otherwise Indy will handle it and take the appropriate actions for you automatically.

9.4.6 Simple Solution

Because the EIdConnClosedGracefully is a common exception especially with certain servers it descends from EIdSilentException. On the Language Exceptions tab of Debugger Options (Tools Menu) you can add EIdSilentException to the list of exceptions to ignore. After this is added the exceptions will still occur in the code and be handled, but the debugger will not stop the program to debug them.

Section

Implementing Protocols

10 Implementing Protocols

Indy implements all the common protocols and many of the lesser used protocols as well. There are still cases that you will need to implement protocols yourself. The most common reason for implementing a protocol is that you need to implement something custom for which no protocol exists.

The first step is to understand the protocol that will be implemented. There are three basic types of protocols:

1. **Standard** - These are protocols that are internet standards. To understand the protocol simply find the RFC's associated with the protocol. The RFC's will detail the protocol in depth.
2. **Custom** - Custom protocols are used when no protocol exists for the desired function. Creating a custom protocol will be covered next.
3. **Vendor** - This refers to protocols which interface with proprietary systems or devices. Vendor protocols are protocols that the vendor developed as a custom protocol and you are now left to interface with. I wish you the best of luck with vendor protocols because most of them are implemented by hardware designers who had one C++ class in college and have no experience in implementing protocols.

Most protocols are conversational and plain text, and unless you have a compelling reason otherwise, all of your protocols should be as well. These two terms are covered in the following sections.

The first step in building a client or server is to understand the protocol. For standard protocols, this accomplished reading the appropriate RFC. For custom protocols, a protocol must be established.

Most protocols are conversational and plain text. Conversational means that a command is given, a status response follows, and possibly data. Protocols that are very limited in scope are often not conversational, but are still usually plain text. Plain text makes protocols much easier to debug, and to interface to using different programming languages and operating systems.

Status codes are traditionally a 3 digit number. There is no standard that defines status codes or their conventions, but many protocols follow a de facto convention of the following:

- 1xx - Informational
- 2xx - Success
- 3xx - Temporary Error
- 4xx - Permanent Error
- 5xx - Internal Error

Each error typically has a unique number associated to it, but some protocols reuse numbers and only provide unique numeric response for each command rather than for the protocol.

10.1 Protocol Terminology

Before discussing protocol implementation some terminology will be presented and explained.

10.1.1 Plain Text

Plain text defines that all commands and replies are in 7 bit ASCII format. While it is quite common to transfer data in responses as binary, nearly all protocol command and replies are plain text. Binary should never be used for command and replies unless an overriding reason can be found.

Using plain text ensures easy debugging and testing. It also ensures inter operability between operating systems and programming languages.

10.1.2 Command

A command is a text string that is sent between a client and a server to request information or that some operation be performed. Examples of commands are HELP, QUIT, LIST, etc.

Commands may also contain optional parameters, separated by spaces.

Example:

```
GET CHARTS.DAT
```

GET is the command, and CHARTS.DAT is a parameter. In this example only one parameter is specified, but commands may have multiple parameters. Normally the parameters are separated by a space very similar to command entered in a DOS window or command shell.

Commands are **always** coded in English. This might seem biased, however a common ground must be chosen and English is the most common language in both the technical world as well as the business world. If English is not used, protocols are not very useful.

One example of localization of commands is in Microsoft Office. Microsoft Office can be automated using OLE automation. However Microsoft localized the names of methods and properties of the objects. This means that if you write an application using Microsoft Office Automation for the US version, your software will not work in France where users likely have the French version of Microsoft Office.

10.1.3 Reply

A reply is a short reply that is sent in response to a command that has been issued. The reply contains status information about the success or failure of the ability to perform the requested command, and sometimes contains small amounts of data as well.

For example if the command *GET customer.dat*, is issued, a reply of *200 OK* might be returned indicating that the command is understood and will be processed. Replies are normally just one line, but can contain multiple lines.

Unlike commands, the textual part of replies may be localized into other languages as long as it conforms to 7 bit ASCII. This is because the protocol itself relies on the numeric portion and the textual portion is for display to the end user or debugging.

10.1.4 Response

A response is the data portion that is returned in response to a command. Responses are optional and do not accompany all commands. Responses occur after the reply has been sent so the sender of the command knows whether to expect a response or not, and the format of the response.

Responses can be textual or binary. If a response is textual it is usually in RFC Response format.

10.1.5 Conversations

Most protocols are conversational. Some simple ones are not, but they are usually still plain text.

A conversation means that the protocol command structure is constructed as follows:

1. Command given
2. Status reply returned
3. Optional data response

Referring back to the postal code lookup demo a conversation looked like this:

```
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

Seperating into the individual pieces of a conversation it looks as follows:

Command:

```
Client: lookup 37642 77056
```

Reply:

```
Server: 200 Ok
```

Response:

```
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

Each of these items is covered in detail in the following sections.

10.2 RFC Definitions

There are no actual standards for the terms defined in this section. However they terms are based on "quasi standards" that nearly all RFC text based protocols adhere to.

One notable exception is POP3. It is a mystery why the designers decided to not only be different than every other RFC. The POP3 protocol is actually more limited and offers no additional functionalty relating to the protocol. It was a completely needless diversion.

IMAP4 which was intended to be the successor to POP3, furthers this trend with further awkward and non standard mechanisms. IMAP4 is not in widespread use, and POP3 remains the standard mail protocol.

Aside from these two notable mentions, the text protocols stick to an established "quasi standard". This "quasi standard" is a uniform way of sending commands, replies, and responses.

10.2.1 RFC Status Codes

RFC status codes are of the form:

```
XXX Status Text
```

Where XXX is numeric from 100-599.

The 3 numeric digits contain the meaning of the reply and are used at run time to detect the outcome of a command. Usually the optional text to follow is for display to the user or for debugging purposes. In such cases the response is usually English but may be localized as long as 7 bit ASCII is used. In some cases where the data is short, data is returned in the optional text field. In those cases the data must still be 7 bit ASCII, but the protocol itself determines the language and formatting restrictions. In most cases it is data and language neutral. An example might be a command "TIME" that returns "15:30" in the optional text field.

An example of an RFC reply might look like this:

```
404 No file exists
```

404 is the numeric response and "No file exists" is the optional text message. Only 404 should be interpreted in this case by the program as 404 would be defined by the protocol to mean exactly this. The text message is merely for debugging, logging, or display to the user. The text message may vary from implementation to implementation and maybe localized to other languages. Languages that cannot properly be displayed in 7 bit ASCII must transliterated to English characters.

The numeric part of the status code can be assigned any meaning. However it is a common convention to follow the following rule:

```
1xx – Informational
2xx – Success
3xx – Temporary Error
4xx – Permanent Error
5xx – Internal Error
```

The numbers are generally unique, but not always. That is if you assign 201 to mean "File not found", many commands may reply with 201 and the meaning is always the same. In some rarer cases, the meaning of the number depends on the command issued. That is, each command assigns specific meanings to each numeric reply.

Numeric codes ending in 00, that is 100, 200, and so on are reserved for general responses that do not have specific meanings associated with them. 200 is often associated simply with "Ok".

Status codes may also be multiline to conain larger text responses. In such cases multiple lines are sent with each line except the last containing a dash after the numeric code instead of a space.

Example multi line status code:

```
400-Unknown Error in critical system
400-The server encountered an error and has no clue what caused it.
400-Please contact Microsoft technical support, or your local
400-tarot card reader who may be more helpful.
400 Thank you for using our products!
```

10.2.1.1 Examples

Here are some example status codes taken from the HTTP protocol. As you can see they correpond to the categorization by the first digit.

```
200 Ok
302 Redirect
404 Page not found
500 Internal Error
```

If you have seen *500 Internal Error*, chances are you were using Microsoft IIS.

10.2.2 RFC Reply

An RFC reply is a reply that is returned as a RFC status code.

10.2.3 RFC Response

An RFC response is a textual response that is terminated by a single period on a line by itself. If the data contains a line that consists of solely a period, the line is converted to a two periods for transmission, and converted back during the receive.

RFC responses are very common when the amount of data to be returned is unknown. It is used by HTTP, Mail, News and other protocols.

Indy methods that support RFC response are Capture (for receiving) and WriteStrings (for sending).

10.2.4 RFC Transactions

A RFC transaction is a conversation that consists of a command, a reply, and an optional response, all in RFC format. Command handlers and other parts of Indy are built around RFC transactions.

Example transaction:

```
GET File.txt
201 File follows
Hello,

  Thank you for your request, however we cannot grant your
request for funds for researching as you put it in your
application "A better mouse trap".

   Thank you, and please do not give up.
.
```

10.3 TIdRFCReply

TIdRFCReply facilitates sending and receiving RFC Replies. TIdRFCReply has three primary properties: NumericCode, Text and TextCode. NumericCode and TextCode are mutually exclusive. TextCode is a property to handle protocols such as POP3 and IMAP4.

To generate a reply, set the NumericCode property (or TextCode) and optionally enter text into the Text property. Text is of type TStrings to allow multi line replies.

TIdRFCReply has methods for writing properly formatted replies, and also for parsing text into a TIdRFCReply instance.

TIdRFCReply is used for sending replies to commands, and also by ReplyException, ReplyUnknown and Greeting properties of TIdTCPServer.

10.4 ReplyTexts

The numeric code in a reply are typically unique for each error. The HTTP protocol for instance uses 404 for "Resource not found". Many different commands are permitted to return 404 as an error, but 404 always signifies the same error. To avoid the duplication of the text for the 404 error each time it is specified, TIdTCPServer also has a ReplyTexts property.

ReplyTexts is a collection of TIdRFCReply instances that can be managed at run time or design time. ReplyTexts is used to maintain a list of texts that correspondes to each numeric code. When a TIdRFCReply is used in a TCPServer that has a numeric code but no text, Indy will look for a corresponding match in ReplyTexts and use its text.

So instead of needing to include the text each time a 404 is required as shown here:

```
ASender.Reply.SetReply(404, 'Resource Not Found');
```

The following code can be used:

```
ASender.Reply.NumericCode := 404;
```

Before Indy actually sends the reply, it will set the reply's Text property from a matching entry in ReplyTexts. This allows all reply texts to be kept in a central location and be managed easily.

10.5 The Chicken or the Egg?

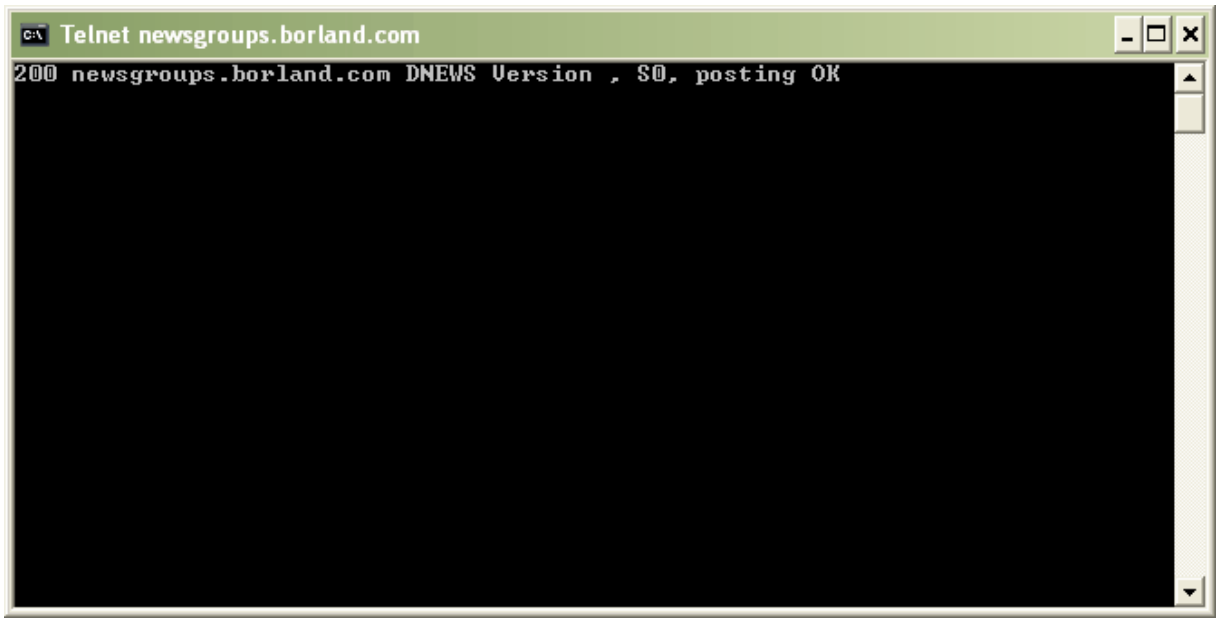
When building systems in which you will build both the client and the server pieces, the followig question will likely arise. "Which one should I build first, the client or the server?" Both are needed for the other one to test against.

The answer is simpler than might be initially thought. The server is easier to build first. To test a client, you need a server. To test a server, you need a client. However since nearly all protocols are text based, a client can easily be simulated using a telnet application.

To test this, connect to a known server on its port. From a command shell type:

```
Telnet newsgroups.borland.com 119
```

Now press enter. After it connects you should see a screen similar to that in the figure below.



Windows 95/98/ME, and NT may look a little bit differently than Windows 2000 or Windows XP, but the results are the same. Other versions of Windows launch telnet as a new application with its own window. The figure above is from Windows XP's telnet client, which is a console application.

The commend "Telnet newsgroups.borland.com 119" instructed the telnet client to connect to the server newsgroups.borland.com on port 119. Port 119 is the port for NNTP (News). Just a telnet client was used to connect to the Borland news server, the telnet clien can be used with any server that uses a text based protocol.

To disconnect from the news server type "Quit" and hit enter.

10.6 Defining a Custom Protocol

The task of a network developer consists not only of interfacing with existing systems, but often of creating completely new ones. In such a case a new protocol will need to be created.

The first step in building a client or server is to understand the protocol. For standard protocols, this accomplished reading the appropriate RFC. If the protocol is not a standard or already defined, one must be defined.

When defining a protocol, the following decisions should be made first:

- Text or binary commands? Unless there is an overriding requirement, use text commands. Text commands are easier to understand and debug.
- TCP or UDP? This really depends on the protocol being built and the needs. Study the characteristics of both and decide carefully. In most cases TCP is the right choice.
- Port - Each server application needs a dedicated port to listen on. Ports below 1024 are reserved and should never be used unless implementing a protocol which has an assigned port below 1024.

Once the questions have been decided the commands, replies and responses need to be designed.

10.7 Peer Simulation

Traditionally the only way to build a client and a server is to build the server first, or build them in paralell. However Indy has a way that you can completely build the client or the server without the other. This allows either one to be built first. In some cases one may need to be built without reasonable access to the other. In such cases peer simulation can be used. Peer simulation covered later in the debugging section.

10.8 Postal Code Protocol

This section will cover the postal code protocol in more detail which was introduced earlier in the postal code client. The server will be presented later.

This project has been designed to be as simple as possible. Postal code (Zip code for American readers) lookup will allow a client to ask a postal code server what city and state a postal code is assigned to.

The sample data that the server uses are US postal codes which are called zip codes. The protocol can handle other postal codes as well, but US zip codes were readily available at the time of demo construction.

For those of you outside the United States who may not be familiar with zip codes, a zip code is a US postal code that specifies a postal delivery area. Zip codes are numeric and five digits in length. Zip codes may also contain an additional four more optional digits in the form 16412-0312. Such types are called Zip+4. The four additional digits specify local delivery information and are not necessary for locating a city.

For the postal code protocol the following decsisions have been made:

- Text commands
- TCP
- Port: 6000. Port 6000 is a port commonly used in Indy for demos. It has no significance.

The postal code protocol will support the following commands

- Help
- Lookup <Post Code 1> <Post Code 2> ...
- Quit

When designing a protocol, its good to become familiar with the corner stone protocols such as NNTP, SMTP, and HTTP and use them as models. Ignore POP3 and IMAP4. These are good examples of how not to design protocols.

Since NNTP supports sending and receiving of messages in one protocol, NNTP will be consulted at various times in this document for your benefit.

10.8.1 Help

Help is a commonly implemented command and has little use for automated clients. Help is useful for human's who are testing or manually interacting with servers. Nearly all servers implement some form of basic help.

Help is useful in determining which commands and possible extension commands that a server

supports.

Here is an example help command response from the Borland news server:

```
help
100 Legal commands
  authinfo user Name|pass Password
  article [MessageID|Number]
  body [MessageID|Number]
  check MessageID
  date
  group newsgroup
  head [MessageID|Number]
  help
  ihave
  last
  list [active|active.times|newsgroups|subscriptions]
  listgroup newsgroup
  mode stream
  mode reader
  newgroups yymmdd hhmmss [GMT] [<distributions>]
  newnews newsgroups yymmdd hhmmss [GMT] [<distributions>]
  next
  post
  slave
  stat [MessageID|Number]
  takethis MessageID
  xgtitle [group_pattern]
  xhdr header [range|MessageID]
  xover [range]
  xpat header range|MessageID pat [morepat...]
.
```

For postal code protocol the server shall respond with the same 100, plus appropriate text.

10.8.2 Lookup

The lookup command accepts one or more postal codes for lookup and returns the cities associated with them. The data is returned in RFC response format. If no matching entry is found it is not returned. The reply for a successful lookup command is "200 Ok".

Example:

```
lookup 37642 16412
200 Ok
37642: CHURCH HILL, TN
16412: EDINBORO, PA
.
```

Even if no matching entries are found, lookup will respond with "200 Ok".

```
lookup 99999
200 Ok
.
```

This is a design decision. If lookup only accepted one parameter it would be better to return 200 if a match was found, and a 4XX error code if no matches were found. Since lookup can return partial valid data, its better to always return 200.

Example of partial valid data return:

```
lookup 37642 99999
```

```
200 Ok
37642: CHURCH HILL, TN
.
```

If the protocol returned an error code, valid data would be ignored. This design decision allows the server to return data that it is able to reference without blocking with an error.

10.8.3 Quit

This quit command is very straightforward. Quit signals that the client is about to end the session and disconnect.

Looking at the Borland news server again it responds like this:

```
quit
205 closing connection - goodbye!
```

```
The postal code protocol responds similarly:
```

```
quit
201-Paka!
201 2 requests processed.
```

The postal code protocol replies with a multi line reply. This is not defined by the protocol itself. Any RFC reply may be a single or multi line reply. Indy parses both types automatically.

Section XI

Proxies

11 Proxies

A common question is "How do I use Indy with a proxy?". While one would expect that an easy answer could be given. Such an answer is not available because there are many different types of proxies. Some protocols even also have their own method of negotiating proxies. The most common kinds of proxies will be explained.

Proxies and firewalls perform similar roles, but have different goals. Because they share similar roles, they can be used to perform each others functions and often are bundled together as combination firewall proxies.

Proxies can be defined into two categories:

- Transparent
- Non Transparent

11.1 Transparent Proxies

Transparent proxies are proxies which require no changes to the protocol which will use the proxy. Transparent proxies often exist without any knowledge of developers or users.

While transparent proxies have no effect on clients, they do affect servers. In most cases servers behind such proxies are hidden from the outside world. To allow servers to be accessed from the other side of the proxy ports must be mapped from the outside to the inside.

11.1.1 IP Masquerading / Network Address Translation (NAT)

An IP masquerading or Network Address Translation (NAT) proxy allows all outbound connections to occur transparently and has no effect on clients. Clients behave normally and do not require any special configuration.

Microsoft Internet Connection Sharing is implemented using this method.

11.1.2 Mapped Ports / Tunnels

A mapped port or tunnel proxy works by creating tunnels across a blocked route. The route may be blocked because of network configuration, may be a bridge between separate networks, or may be an intentional block to protect or firewall the inner network.

The inner network is defined as the side of the network which the local network exists, and the outer network is defined as the network or external network that is the target of the inner network.

Because of the blocked route, all access must be directed through mapped ports. Ports are assigned on a local server that has access to the outside world. This server then passes data back and forth between the outer and inner network. The disadvantage of mapped ports is that each port is mapped to a fixed remote host and port. For protocols such as mail and news which the servers can be determined before hand this works quite well. However for protocols like HTTP this method cannot be employed because the remote locations cannot be known before hand.

Mapped ports can also be used to map ports from the outer network to the inner network. Mapped ports are often used in conjuction with NAT proxies to expose servers to the outer network.

11.1.3 FTP User@Site Proxy

There are several methods to implement an FTP proxy. The most common FTP proxy type is called User@site.

Using the User@Site method, all FTP sessions are connected to a local proxy server. The proxy pretends to be a FTP server. The proxy server intercepts and interprets the FTP requests. When the proxy asks for user name instead of sending just the user name, the user name and desired FTP server are sent as the user name in the form username@ftpsite. The proxy then connects to the desired FTP site and intercepts the transfer commands.

For each transfer command, the proxy dynamically maps a local port for the data transfer, and modifies the transfer information returned to the client. The FTP client the contacts the proxy instead of accessing the real FTP server directly. Because of the translations, the FTP client does not know that a proxy is even involved.

For example given an FTP site of ftp.atozedsoftware.com, a username of joe and a password of smith, a normal FTP session would be configured like this:

```
Host: ftp.atozedsoftware.com
User: joe
Password: smith
```

If a User@Site proxy exists, and the host name proxy name is corpproxy, the FTP session would be configured like this:

```
Host: corpproxy
User: joe@ftp.atozedsoftware.com
Password: smith
```

11.2 Non Transparent Proxies

Non transparent proxies require changes to the protocols which will use them. Many protocols have provisions for non transparent protocol specific proxies.

11.2.1 SOCKS

SOCKS is a proxy that does not require any changes to the higher level protocol, but works on the TCP level. For protocols to use a SOCKS proxy, the software must implement it at the TCP level.

If software does not explicitly support the SOCKS proxy, it cannot be used with a SOCKS firewall. A lot of popular software such as browsers and ICQ support SOCKS, but the majority of software does not. Because of this, SOCKS must often be deployed in conjunction with internal servers, mapped ports, other proxies, or a combination.

For software to support SOCKS, instead of connecting to the destination server it first connects to the SOCKS proxy. It then passes over a record of data containing the destination server and optional proxy authentication credentials. The SOCKS server then dynamically tunnels the connection to the server.

Because the SOCKS protocol works dynamically on the record passed during connection, SOCKS is highly configurable and very flexible.

11.2.2 HTTP (CERN)

An HTTP proxy, sometimes referred to as a CERN proxy, is a specialized proxy that only proxies browser traffic. In addition to HTTP it can also proxy FTP traffic, if the FTP is done over HTTP. HTTP proxies can also provide caching and are often used for this purpose alone.

Many corporate environments have completely firewalled off their inner networks and only allow access to the outside world using HTTP and mail. Mail is provided using internal mail servers and HTTP is provided using a HTTP proxy. Because of this HTTP has become the "Ultimate firewall friendly" protocol and is why so many newer protocols can use HTTP as a transport. SOAP and web services are notable examples.

Section XII

IOHandlers

12 IOHandlers

Indy is customizable and extensible in many ways without the requirement to modify the Indy source code directly. One example of this extensibility is IOHandlers. IOHandlers allow you to use any I/O source with Indy. IOHandlers should be used when you wish to use an alternate I/O mechanism or create a new transport mechanism.

IOHandlers perform all the I/O (Input / Output) for Indy. Indy no longer performs any of its own I/O outside of an IOHandler.
An IOHandler is used to send and receive raw TCP data for Indy components.

IOHandlers allow classes to be defined that handle the input/output for Indy. Normally all I/O is done via the socket and handled by the default IOHandler, TIdIOHandlerSocket.

Each TCP client in Indy has a IOHandler property that can be assigned to an IOHandler, as does each server connection. If an IOHandler is not specified, an implicit instance of a TIdIOHandlerSocket will be created automatically and used by the TCP client. TIdIOHandlerSocket implements I/O using a TCP socket. Indy includes additional IOHanders: TIdIOHandlerStream and TIdSSLIOHandlerSocket as well.

Other IOHandlers can be created to allow Indy to use just about any I/O source that you can imagine. Currently Indy only supports sockets, streams and SSL as I/O sources, but IOHandlers allow for many other possibilities as well. While there are no current plans, IOHandlers could be implemented to support Tunneling, IPX/SPX, RS-232, USB or Firewire. Indy does not restrict your I/O choices, and by using IOHandlers Indy allows you to choose any I/O that you require.

12.1 IOHandler Components

12.1.1 TIdIOHandlerSocket

TIdIOHandlerSocket is the default IOHandler. If no IOHandler is specified an implicit TIdIOHandlerSocket will be created for use. TIdIOHandlerSocket handles all the I/O related to a TCP socket.

Normally TIdIOHandlerSocket is not use explicitly unless advanced options are needed.

12.1.2 TIdIOHandlerStream

TIdIOHandlerStream is used for debugging and box testing. Using it, live TCP sessions can be recorded as they interact with servers. Later as part of a box, the session can be "replayed". The Indy components do not know that they are not conversing with a real server using an actual connection.

This is a very powerful debugging tool in addition to a QA testing tool. If a customer has a problem, a special build can be sent, or a debug feature turned on to log the session. Using the log files, you can then reconstruct the customer's session in a local debug environment.

12.1.3 TIdSSLIOHandlerSocket

TIdSSLIOHandlerSocket is used to implement SSL support. Normally encryption and compression handlers would be implemented using Intercepts instead of IOHandlers. However, the SSL library that Indy uses (OpenSSL) accepts a socket handle and does the socket communication in the library instead of translating data received and sent by Indy. Because of this, it is implemented as an IOHandler because the library does handle the I/O. TIdSSLIOHandlerSocket is a descendant of TIdIOHandlerSocket.

12.2 Demo - Speed Debugger

Speed Debugger demonstrates how to simulate slow connections. This is useful for both debugging and simulating slower network connections such as modems to test performance of your application.

Speed Debugger consists of a main form and a custom IOHandler. Speed Debugger uses a mapped port component to proxy HTTP traffic for a specified web server. A browser then connects to Speed Debugger and Speed Debugger retrieves a web page from the specified web server, but returns it to the web browser by throttling (slowing down) the data at the specified speed.

The text box is used to specify a web server. Note that it does not specify a URL and therefore does not contain http:// or a web page reference. It only specifies the server host name or IP address. If you have a local web server running you can change it to use 127.0.0.1 to use the local web server.

The combo box is used to select the speed to restrict the connection to and consists of the following choices. The simulated restricted speed is displayed in parentheses.

- Apache (Unlimited)
- Dial Up (28.8k baud)
- IBM PC XT (9600 baud)
- Commodore 64 (2400 baud)
- Microsoft IIS on a PIII-750 & 1GB RAM (300 baud)

When the Test button is pressed Speed Debugger will launch the default browser with the URL http://127.0.0.1:8081/. This will cause the browser to make its requests from Speed Debugger. Speed Debugger listens on port 8081 to avoid conflicting with any existing local web servers.

Speed Debugger can be downloaded from the Indy Demo Playground.

12.2.1 Custom IOHandler

The work of Speed Debugger is done by a custom IOHandler. The mapped port component has a OnConnect event, and this event is used to hook our IOHandler for each outbound client that the mapped port creates. The even looks like this:

```
procedure TformMain.IdMappedPortTCP1Connect(AThread:  TIdMappedPortThread);
var
  LClient: TIdTCPConnection;
  LDebugger: TMyDebugger;
begin
  LClient := AThread.OutboundClient;
  LDebugger := TMyDebugger.Create(LClient);
  LDebugger.BytesPerSecond := GSpeed;
  LClient.IOHandler := LDebugger;
end;
```

The custom IOHandler class is named TMyDebugger and is implemented by descending from the default TCP socket IOHandler TIdIOHandlerSocket. Since TIdIOHandlerSocket already implements all of the actual I/O, TMyDebugger only needs to slow the data transfer down to the specified speed. This is done by overriding its .Recv method.

From the .Recv method the inherited .Recv is called to receive data. Then based on the specified speed limit, the appropriate delay is calculated. If the calculated delay is larger than the actual time the inherited .Recv call took to execute, a call to Sleep is made with the difference. It might sound complicated, but its really simple. The .Recv method appears below.

```
function TMyDebugger.Recv(var ABuf; ALen: integer): integer;
var
  LWaitTime: Cardinal;
  LRecvTime: Cardinal;
begin
  if FBytesPerSecond > 0 then begin
    LRecvTime := IdGlobal.GetTickCount;
    Result := inherited Recv(ABuf, ALen);
    LRecvTime := GetTickDiff(LRecvTime, IdGlobal.GetTickCount);
    LWaitTime := (Result * 1000) div FBytesPerSecond;
    if LWaitTime > LRecvTime then begin
      IdGlobal.Sleep(LWaitTime – LRecvTime);
    end;
  end else begin
    Result := inherited Recv(ABuf, ALen);
  end;
end;
```

Section XIII

Intercepts

13 Intercepts

An Intercept is at a higher level than an IOHandler and is used to modify or capture data independent of its source or destination. Intercepts are used for logging, debugging, and encryption. Other possible uses are compression or statistical analysis.

Intercepts have been drastically changed in Indy 9.0. In Indy 8.0 Intercepts could provide for some very limited functionality of an IOHandler. Intercepts could transform already received data or data before sending. Intercepts no longer support any IOHandler functionality as the new IOHandler classes handle all of this functionality.

Intercepts can still perform data transformation and in a much more flexible way than in Indy 9.0. Indy 8.0's transformation capabilities had many limitations, such as the data was required to remain the same size. This made it impossible for compression to be implemented as an intercept and limited intercepts to logging and compression that did not change the data size.

Intercepts work by allowing data to be modified after it has been received from an IOHandler, or be modified before it is sent to an IOHandler. Intercepts are currently used to implement logging and debugging components. Intercepts could also be used to implement encryption, compression, statistic collectors, or bandwidth limiters.

13.1 Intercepts

Intercepts intercept inbound and outbound data and allow it to be logged or modified. Intercepts allow incoming data to be modified after it has been read from the network but before returned to the user. Intercepts also allow outbound data to be modified after it has been received by the user, but before it is transmitted. Intercepts can be used to implement logging, encryption, and compression.

Client intercepts are connection based and are per connection. They can also be used with server if assigned to individual connections.

Note that Intercepts in Indy 9 differ from Intercepts in Indy 8. In Indy 8, Intercepts performed a combined role of Intercept and IOHandler. This made it difficult to have separate Intercept and IOHandler functionality. Indy 8's Intercepts also could not change the size of the data and thus could not perform compression.

13.2 Logging

Indy 8.0 had one log component that could use different destinations. Indy 9.0's logging components are now based on a new common logging class and are specialized logging classes. The common logging class also introduces properties and functionality such as being able to log the time in addition to the data.

All logging classes are implemented as intercepts. This means that they catch incoming data after it has been read from the input source and before outgoing data is written to the output.

The specialized logging classes are:

- TIdLogEvent - TIdLogEvent fires events when data is sent, received, or a status message occurs. TIdLogEvent is useful for implementing custom logging without needing to implement a new logging

class.

- TIdLogFile - Logs data to a data file.
- TIdLogDebug - Logs data to the debug window in Windows and to the console in Linux. It also marks the data as received data, sent data or status information. TidLogDebug is useful for simple debugging.
- TIdLogStream - Does not comment or mark up the data as the other logging classes do. Instead it merely writes raw data to the specified streams. TIdLogStream has many uses, but it is extremely useful for QA testing, and remote debugging.

Custom log classes can also be built.

Section XIV

Debugging

14 Debugging

Clients are generally quite easy to debug especially when compared to servers. Clients only handle a single connection and can usually be debugged using normal debugging techniques. In this section a few useful tips will be examined that are useful for both clients and servers.

14.1 Logging

A really easy way to see what is going on with a client without tracing through the code is to use the TIdLogDebug or the TIdLogFile. TIdLogDebug will log directly to the debug window and is very handy for watching the data that the client is sending and receiving in real time. If you do not need to watch the traffic in real time, use a TIdLogFile. After the client is finished you can then look at the contents of the file to see what occurred during the session.

14.2 Peer Simulation

There may exist a time that you need to simulate a client or a server that is not accessible. It may be for security, bandwidth or other reasons. In such cases you can use peer simulation. Peer simulation can also be used to build a client before building a server, or create test scripts for servers by simulating the client.

Peer simulations are performed by using a TIdIOHandlerStream and assigning the output stream to a text file, and leaving the input stream nil. This will instruct the TIdIOHandlerStream to read all of the data to send to the peer from the text file, and ignore all data returned from the peer.

If you assign the input stream, it will log the data from the peer.

14.3 Record and Replay

One really useful feature is the ability to record sessions and replay them later. This is extremely useful for both regression testing, but also remote debugging. If you have a customer in a remote location you can send them a special build or have the turn on an option in your software to record a session. They can then send you the recorded file and you can simulate their client or server, without ever needing to actually connect to their server.

To accomplish this use a TIdLogStream to record the received data in a file. You can also record the data that the client sent in a separate file, but you will not need this unless you wish to look at it manually. When you receive the file you can attach a TIdIOHandlerStream component to the client.

IOHandlers also have another really neat use. That use is record and reply. A live session can be recorded using the log components, and later replayed using stream IOHandlers. Imagine that you have a customer that is having trouble, but you cannot reproduce their problems and cannot be on site. You can have them log a complete session to a set of files, and then using stream IOHandlers completely replay the session on your development machine. The Indy team is starting to use these as part of their QA process and box test suite. I plan to cover this in a future installment.

Section XV

Concurrency

15 Concurrency

In a threaded environment resources must be protected so that resources are not corrupted by allowing access by more than one thread at a given time.

Concurrency and threads are intertwined, and choosing which to learn first can be difficult. This text will cover concurrency issues first which will build proper understanding for learning threads.

15.1 Terminology

15.1.1 Concurrency

Concurrency is the state of many tasks occurring at the same time. When concurrency is implemented properly it might be considered "harmony". When implemented poorly, "chaos".

In most cases, a task is a thread. However tasks can also be processes or fibers.

The line dividing the two is often a fine one, and usage of proper techniques is the key.

15.1.2 Contention

What exactly is contention? Contention is when more than one task tries access to a single resource at the same time.

Those of you who grew up in a large family definitely understand contention, and maybe an example will explain concisely. Imagine what happens in family of six children when Mom puts a small pizza on the table for dinner. That is contention.

Whenever multiple concurrent tasks need to access data in a read/write fashion access to the data must be controlled to protect its integrity. If access is not controlled, two or more tasks may "crash" when one tries to read a variable while another tries to write to it simultaneously. If one task is writing while another is reading, the one reading may retrieve partially written data and receive corrupt data. Typically this will not cause any exception and instead will simply cause errors later in the program.

Contention problems often do not occur in low volume implementations and thus often do not occur during development. Because of this proper technique and load testing should be performed during development. Otherwise it is a bit like playing Russian Roulette, and problems will only occur randomly during development, but frequently during deployment.

15.1.3 Resource Protection

Resource protection is the method to prevent the problems caused by contention. Resource protection functions by allowing only one task at a given time to access a specified resource.

15.2 Resolving Contention

Whenever multiple threads need to access data in a read/write fashion access to the data must be controlled to protect its integrity. This can be intimidating to programmers not familiar with threading. However, most servers do not require global data. Those that do typically need only to read the data after it has been initialized during program start up. As long as there is no write access, threads can read global data with no side effects.

The common ways of resolving contention are covered next.

15.2.1 Read Only

The simplest method is read only. Any simple type (integers, strings, memory) that is used in a read only capacity does not require any protection. This also extends to many complex types such as TLists, etc. Classes are safe in read only capacity if they do not make use of any global or field variables in a read / write capacity.

In addition, resources can be written to before any possible reads. This allows initialization of resources on startup, before the tasks that access it are started.

15.2.2 Atomic Operations

There is a methodology that says that with atomic operations that resources need not be protected. An atomic operation is an operation that is too small to be divided up by the computers processor. Because of its small size it will not suffer from contention since it will execute by itself and not be task switched during its execution. Typically atomic operations are lines of source code that compile into a single assembly instruction.

Typically tasks such as reading or writing to an integer or boolean field are considered atomic operations as they compile to a single move instruction. However it is my recommendation that you never rely on atomic operations because in some cases even a write to a integer or boolean can involved more than a single operation depending on where the data is read from first. In addition, this relies on internal compiler knowledge which is subject to change without notice to you. Relying on atomic operations at the source code level will produce code that will be problematic in the future and may act quite differently on multi processor machines or other operating systems.

I have seen atomic operations defended strongly. However an very prominent upcoming issue that proves my point is .net. When your source code is first compiled to IL, and then recompiled to machine code possibility on different platforms my different vendors, can you honestly be sure your line of source code will be an atomic operation in the end?

The choice is yours and there are certainly arguments both in favor of and against atomic operations. Relying on atomic operations saves only a few microseconds in most cases, and a few bytes of code. I strongly recommend against atomic operations as they provide such little benefit and have huge liabilities. Treat all operations as if they were not atomic operations.

15.2.3 Operating System Support

Many operating system have support for very basic threadsafe operations.

Windows supports a set of functions known as Interlocked functions. The usefulness of the functions is quite limited and consists of simple functionality on integers such as increment, decrement, add, swap and swap-compare.

The number of functions varies with different editions of Windows, and can cause deadlocks in older versions of Windows. In most applications they offer little performance benefit.

Because of the combined factors of limited use, varying support, and little performance benefit, it is advised to use the Indy threadsafe counterparts instead.

Windows also contains support for special IPC (interprocess communication) objects that have Delphi wrappers. These objects are extremely useful for threading as well as IPC.

15.2.4 Explicit Protection

Explicit protection involves each task knowing that a resource is protected and taking explicit steps before accessing a given resource. Normally such code is in a single routine which is executed by many tasks concurrently, or is encapsulated into a routine that is called from many different locations and acts as a threadsafe wrapper.

Explicit protection normally takes use of a resource protection object. In short a resource protection object limits access to a resource to one task at a time. A resource protection object does not actually limit access to a resource, if it did it would have to know specifics about each and every resource type. Instead they act like a traffic signal and code is adapted to obey and provide input to the traffic signal. Each resource protection object implements a different type of traffic signal using different logic, different inputs, and varying levels of overhead. This allows for a different resource protection

object to be chosen to better fit different types of resources and situations.

Resource protection objects exist in several forms and are covered individually later.

15.2.4.1 Critical Sections

Critical sections can be used to control access to global resources. Critical sections are lightweight and are implemented in the VCL at TCriticalSection. In short, a critical section allows a single thread in a multi-threaded application to temporarily block the execution of all other threads attempting to use the same critical section. Critical sections work like a traffic light that only turns green when the ahead roadway is clear of all vehicles. Critical sections can be used to assure the only one thread at any given time is executing a block of code. Because of this, blocks protected by critical sections should be kept as small as possible as they can severely impact performance if not properly used. Because of this, each unique block should also use its own TCriticalSection and instead of reusing a single application wide TCriticalSection.

To enter a critical section the Enter method is called and Leave is used to leave the critical section. TCriticalSection also has Acquire and Release methods that do the same exact thing as Enter and Leave respectively.

Imagine a server that needs to log information about logged in clients and display the information in the main thread. One option would be to use synchronize. However using this method will negatively impact the performance of the connection threads if many clients log in at the same time. Depending on the needs of the server, a better option may be log the information and have the main thread read the information using a timer. The following code is an example of this technique that utilizes a critical section.

```
var
  GLogCS: TCriticalSection;
  GUserLog: TStringList;

procedure TformMain.IdTCPServer1Connect(AThread: TIdPeerThread);
var
  s: string;
begin
  // Username
  s := ReadLn;
  GLogCS.Enter; try
    GUserLog.Add('User logged in: ' + s);
  finally GLogCS.Leave; end;
end;

procedure TformMain.Timer1Timer(Sender: TObject);
begin
  GLogCS.Enter; try
    listbox1.Items.AddStrings(GUserLog);
    GUserLog.Clear;
  finally GLogCS.Leave; end;
end;

initialization
  GLogCS := TCriticalSection.Create;
  GUserLog := TStringList.Create;
finalization
  FreeAndNil(GUserLog);
  FreeAndNil(GLogCS);
end.
```

In the Connect event the username is read into a temporary variable before entering the critical section. This is done to avoid the possibility of a slow client blocking the code in the critical section. This allows the network communication to be performed before entering the critical section. To keep performance at a maximum, the code in the critical section is kept to an absolute minimum.

The Timer1Timer event is fired in the main thread by a timer on the main form. The interval of the timer can be shortened to provide for a more frequent update, but can potentially slow down the acceptance of connections. If the log functionality is extended into other portions of the server in addition to just logging the connection of the user, the greater the potential for a bottleneck. The greater the interval, the less real time the update of the user interface. However many servers have no interface at all, and the ones that do, the interface is normally secondary and a much lower priority than servicing clients so this is a perfectly acceptable trade off.

15.2.4.1.1 Note to Delphi 4 Standard Users

TCriticalSection is in the SyncObjs unit. The SyncObjs unit is not included in Delphi 4 Standard Edition. If you are using Delphi 4 Standard Edition there is a SyncObjs.pas file available at the Indy website that does not implement everything in Borland's SyncObjs.pas, but does implement the TCriticalSection class.

15.2.4.2 TMultiReadExclusiveWriteSynchronizer (TMREWS)

In the previous example, TCriticalSection has been used to protect access to global data. In those cases the global data was always updated. However, if global data is being accessed sometimes as read only, use of a TMultiReadExclusiveWriteSynchronizer may produce more efficient source code. TMultiReadExclusiveWriteSynchronizer is a long and hard to read class. Because of this it will simply be referred to as TMREWS.

The advantage of using a TMREWS is that it allows for concurrent reading by multiple threads, while acting like a critical section and allowing only one thread access during reads. The disadvantage is the TMREWS is more expensive to use.

Instead of Enter/Acquire and Leave/Release, TMREWS has the methods BeginRead, EndRead, BeginWrite and EndWrite.

15.2.4.2.1 Special Notes on TMREWS

Prior to Delphi 6 TMultiReadExclusiveWriteSynchronizer has a problem whereby during a promotion from a read lock to a write lock could cause a dead lock. Because of this you should never use this feature of promoting a read lock to a write lock even though it is documented as being able to do so.

If you need this functionality there is a work around. The work around is to release the read lock and then obtain a write lock. However once you have obtained the write lock you then must again check the condition that first caused you to desire a write lock. If it still exists perform what is needed, otherwise simply release the write lock immediately.

TMultiReadExclusiveWriteSynchronizer also has special considerations when used in Delphi 6. All versions of TMultiReadExclusiveWriteSynchronizer including the one in both update pack 1 and update pack 2 have a serious issue which can cause dead locks. There are no known workarounds. Borland is aware of the issue and has released unofficial patches and is expected to issue official patches.

15.2.4.2.2 TMREWS in Kylix

TMultiReadExclusiveWriteSynchronizer in Kylix 1 and Kylix 2 is implemented internally using a Critical Section and will not offer any advantage over using a Critical Section. It is however included so that code can be written for both Linux and Windows. In future versions of Kylix, TMultiReadExclusiveWriteSynchronizer will probably be upgraded to perform as it does under Windows.

15.2.4.3 Choosing Between Critical Sections and TMREWS

Because the TMREWS has been plagued by problems, my advise is simply to avoid it. If you do decide to use it, you should make sure that it is actually the better option and you have obtained a patched version which is not prone to deadlock behavior.

Proper use of a TCriticalSection in most cases will yield nearly as fast results, and in some cases faster results. Learn to optimize your TCriticalSection's where necessary as improper use of a TCriticalSection will significantly negatively impact performance.

They key to any resource protection is to use multiple resource controllers and keep lock sections small. When such situations can be established a critical section should always be used instead as it is lighter weight and faster than a TMREWS. In general, always use critical sections unless you can explicitly justify the user of a TMREWS.

The TMREWS class works better when the following criteria are met:

1. Access consists of reading and writing.
2. Reading access is the majority.
3. The period of time which the lock must be maintained is extended and cannot be broken into smaller separate pieces.
4. A TMREWS class is available that has been properly patched, or is known to work properly.

15.2.4.4 Performance Comparison

As mentioned prior critical sections are lighter weight and thus faster. Critical sections are implemented by the operating system. The operating system implements them using very fast and small assembly instructions.

The TMREWS class is more complicated and thus has more overhead. It must manage lists of requesters to properly manage the dual state locking mechanism.

To demonstrate the difference a sample project named ConcurrencySpeed.dpr has been created. It performs three simple measurements of the following:

1. TCriticalSection – Enter and Leave
2. TMREWS – BeginRead and EndRead
3. TMREWS – BeginWrite and EndWrite

It performs these tests by running them inside a counter loop for a specified amount of times. For testing purposes it defaults to 100,000 iterations. In my tests the following results were yielded (in milliseconds):

```
TCriticalSection: 20
TMREWS (Read Lock): 150
TMREWS (Write Lock): 401
```

Naturally the measurements are machine specific. However the differences between them is what is important here, not the actual numbers. It can be seen that optimally a read lock on a TMREWS is 7.5 times as slow as a critical section, and that a write lock is 20 times as slow.

It should also be noted that while critical section only has one measurement, TMREWS performance degrades under concurrent usage. The tests performed here were simply in a loop and no other requesters were requesting or had existing locks for the TMREWS to deal with. In a real situation the TMREWS would be even slower than the optimal numbers shown here.

15.2.4.5 Mutexes

Mutexes function almost identically to critical sections. The difference with a mutex is that it is a more powerful version of a critical section with more features, and thus more overhead as well.

Mutexes have the additional capabilities of being named, assigned security attributes, and being accessible across processes.

Mutexes can be used for communication between threads, but rarely are needed. Mutexes are designed and normally used for communication between processes.

15.2.4.6 Semaphores

A semaphore is similar to a mutex, but instead of allowing only one entrant, it allows multiple entrants. The number of entrants that it allows is specified when the semaphore is created.

Imagine that a mutex is a security guard who is guarding access to a bank cash dispensing machine (ATM). Only one person at a time may use it, and the security guard is protecting the machine from the line of people who wish to all use it at the same time.

In this case, a semaphore would be relevant if 4 ATM's were installed. In such a case, the security guard would allow 4 people at a time to enter and use the ATMs, but not more than 4 at a given time.

15.2.4.7 Events

Events are signals that can be used between threads or processes to notify that something has occured. Events can be used to notify other tasks when something has completed, or needs action.

15.2.5 Thread Safe Classes

Thread safe classes are classes specifically designed to protect a specific type of resource. Thread safe classes each implement a specific type of resource and have intimate knowledge of both what the resource is, and how it functions.

Thread safe classes may be as simple as a thread safe integer, or as complex as a thread safe database. Thread safe classes use thread safe objects internally to accomplish their function.

15.2.6 Compartmentalization

Compartmentalization is the process of isolating data and assigning it to be used only by a single task. With servers compartmentalization is often natural as each client can be handled by a dedicated thread.

When compartmentalization is not natural it should be evaluated to see if it is possible. Compartmentalization is often possible by making copies of global data, working on that data, and then returning the results to the global area. By using compartmentalization, data is locked only during initialization and after the task has finished, or during batched updates.

Section

Threads

16 Threads

Threading intimidates many programmers, and often frustrates programmers who are new to threading. Threading is an elegant way to solve many issues and once mastered will become a powerful new technique in your skill set. The topic of threading could easily fill a book of its own.

16.1 What is a Thread?

A thread is a separate preemptive execution path. Using threads allows for different execution paths to be executed simultaneously.

Imagine your computer is company with only one telephone line. Because there is only one telephone line, only one person can use it at a time. However if you install multiple telephone lines others can make telephone calls without being prevented by someone else already using it. Threads allow your application to do more than one thing at a time.

Threading is available even if you only have one CPU. In reality only one thread is executing at a time, however the operating system preemptively interrupts the thread and switches to another one. Each thread is only executed for a very short time each pass. This allows for tens of thousands of "slices" to occur per second. Because the switching is preemptive and unpredictable, to the software the threads appear to be executing in parallel, and software must take precautions about this.

In multiple CPU systems threads can actually execute in parallel, however each CPU can in reality still only execute one thread at a time.

16.2 Threading Advantages

The use of threading provides several advantages over non threaded designs.

16.2.1 Prioritization

Individual threads priorities can be adjusted. This allows individual server connections or threaded clients to be given more or less CPU time.

If you raise the priority of all threads there will not be much of an effect as they will still all be the same priority. They may however gain time of the threads of other processes. You should be careful when you set thread priorities not to set them too high otherwise they can interfere with threads that are dealing with hardware input and output.

In most cases when thread priorities are adjusted instead of raising the priority, you will lower it. This is to allow less important tasks to yield more time to more important ones.

Thread priorities can also be useful in servers and in some cases you may wish to adjust a thread priority based on login. If an administrator or CEO logs in you may wish to increase their thread priority over the rest of the users.

16.2.2 Encapselation

The use to threads allow each task to be self contained and thus less likely to interfere with other tasks and associated data.

If you have ever used Windows 3.1 you probably remember how one misbehaving application could easy bring down the whole system. Threads prevent this. But it also goes farther than this, directly into programming. Without threads, all tasks would have to be accomplished within the same paths of code, creating extra complexities. With threading, each task can be separated into separate independent sections making your code easier to program when tasks must be executed simultaneously.

16.2.3 Security

Each thread can have its own individual security attributes based on authentication or other criteria. This is especially useful in server implementations where each user has a specific thread associated with their connection. This allows the operating system to implement proper security for the user, properly restricting their access to files and other system objects. Without this feature you would have to reimplement security, quite possibly leaving open disastrous security holes.

16.2.4 Multiple Processors

The use of threading will automatically utilize multiple processors if available. If threading is not utilized your application will have only one thread, the main thread. A thread can only be executed by one CPU at a time thus your application will not execute as fast as possible.

Other processes will use the other CPU's however as will the operating system. Certain calls that are made to the operating system by your program are multi threaded internally and your application will receive a small boost from that. In addition the time available for your application will be greater because the other CPU's can handle other applications and lessen the load per CPU.

The best way to take advantage of multiple CPU's however is to multi thread your application. This will not only allow your application to be properly distributed across the CPU's, but will give more CPU time to your application because it contains more threads.

16.2.5 No Serialization

Threading provides true concurrency. Without threading all requests must be handled by a single thread. For this to work each task to be performed must be broken up into small pieces that can always execute quickly. If any task part blocks or takes time to execute, all other task parts will be delayed until it is complete. After each task part is complete, the next one is processed, and so on.

With threading, each task can be programmed as a complete task and the operating system will divide CPU time among the tasks.

16.3 Processes vs. Threads

Processes are different than threads but often confused with processes. A process is a complete new instance of an application which includes all overhead required to run an executable including operating system management overhead and extra memory. An advantage of processes is that they are completely isolated from each other while threads are only partially isolated from each other. If a process crashes, other processes remain unaffected.

16.4 Threads vs. Processes

Threads like processes are independent preemptive execution paths. Threads however run as part of a parent process. Each thread has its own stack, but it shares a common heap with other threads in the same process. Threads are faster to create than processes. Threads also have lower overhead in terms of operating system management and the amount of memory they require to operate.

Because threads are not completely isolated from each other, communication between them is also rather easy.

16.5 Thread Variables

Thread variables are declared using the keyword ThreadVar.

Thread variables are similar to globals and are declared in a similar fashion. The difference is that a global would be global to all threads, while a threadvar will be global to all code, but specific to each thread. That is each thread defines its own "global space".

Thread variables can be useful when it is difficult to pass references to object between libraries or isolated parts of code. However thread variables also have limitations. Thread variables cannot be used or declared inside of packages. Whenever possible, member variables in thread classes should be used instead. They incur lower overhead and are available for use in packages.

16.6 Threadable and ThreadSafe

The term threadsafe is often misused or applied incorrectly. It often refers to both threadable and threadsafe which can lead to confusion and errors. In this text, threadable and threadsafe are strictly defined and refer to separate meanings.

16.6.1 Threadable

Threadable means that an item can be used inside of a thread or used by threads if properly protected using resource protection. The item marked at threadable generaly does not have knowledge of threads in most cases.

If an item is threadable it means that it can be used by a single thread at a time. This can be accomplished by making it local to a thread in most cases, or as a resource protected global item.

Common examples of threadable items are Integer, String, other ordinal types, TList, TStringList, as well as most non visual classes.

The item cannot make unrestricted use of global variables or directly access GUI controls. The unrestricted (and often unnecessary) use of global variables is the most common reason that components are not threadable.

An item may be a library, component, procedure, or other.

16.6.2 Threadsafe

Threadsafe means that the item has explicit knowledge of threads and provides its own resource protection. Threadsafe items can be used by a single or multiple threads without any resource protection.

Examples of threadsafe classes are the VCL's TThreadList, as well as all of Indy's threadsafe classes. Certain operating system objects are also threadsafe.

16.7 Synchronization

Synchronization is the process of passing information from secondary threads to the main thread. The VCL supports this by use of the Sychronize method in TThread.

16.8 TThread

TThread is the threading class that is included in the VCL and provides a good implementation upon which to build threads.

To implement a thread a class is descended from TThread and the Execute method is overriden.

16.9 TThreadList

TThreadList is a threadsafe implementation of TList. TList can be used by any number of threads without the need to protect it from simultaneous access.

TThreadList works similar to TList, but not exactly the same. Some methods such as Add, Clear, and Remove are the same. However for other operations the TThreadList must first be locked using the LockList method. The LockList method is a function and returns direct access to the internal TList. While it is Locked, all other threads will be locked. Because of this it is important to Unlock it again as soon as possible.

Example of operation on items of a TThreadList:

```
with MyThreadList.LockList do try
  for i := 0 to Count - 1 do begin
    // Operate on list items
    Items[i] := Uppercase(Items[i]);
  end;
finally MyThreadList.UnlockList; end;
```

It is very important that the list always be unlocked when the code is finished with it and thus it should always be locked and unlocked using a try..finally construct. If a list is left locked, it will cause a deadlock with other threads attempting access to the list.

16.10 Indy

Indy contains many additional classes that complement the VCL to facilitate threading. These classes are actually independent of the core Indy library and are useful in non-Indy applications as well. They exist in Indy because Indy is designed with threads in mind. Indy not only uses these classes for server implementation but provides them for developer use as well. This section will provide a brief overview of these classes.

16.11 TIdThread

TIdThread is a descendant of TThread and adds more advanced features as well as providing features more suited to use in servers, and also provides support for thread pooling and reuse.

If you are familiar with the VCL's TThread, it is very important to take note that TIdThread differs sharply in a few key areas. With TThread, the Execute method is overridden by descendants, but with TIdThread the Run method is overridden instead. Be sure not to override TIdThread's Execute

method, as it will interfere with TIdThread's internal operations.

For all TIdThread descendants, the Run method must be overridden. When the thread becomes active, Run will be executed. Run will be continually called by TIdThread until the thread has terminated. The use of this may not be readily apparent for most clients, however it is especially useful in nearly all servers and some clients as well. This is also different than TThread's Execute method as the Execute method executes only once and when it exits it is not recalled. When Execute exits, the thread is done.

There are also other differences between TIdThread and TThread, but these are the primary differences, and Run and Execute is the largest.

16.12 TIdThreadComponent

TIdThreadComponent is a component that allows you to build new threads visually by simply adding event at design time. It is basically just a visual encapsulation of TIdThread, but it makes making new threads really easy.

To use a TIdThreadComponent add one to your form, define the OnRun event and set the Active property. An example of TIdThreadComponent can be seen in the TIdThreadComponent demo available at the Indy Portal.

16.13 TIdSync

TThread has a Synchronize method, but it does not support the ability to pass parameters to synchronized methods. TIdSync allows for synchronizations with the ability to pass parameters to the synchronized methods as well. TIdSync also allows for return values to be returned from the main thread.

16.14 TIdNotify

Synchronizations are fine when the number of threads is small. However for servers or applications with many threads, synchronizations become a bottleneck and have a drastic negative impact on performance. To solve this, notifcations can be used instead. Indy's TIdNotify implements notifications. Notifications allows communication with the primary thread, but unlike synchronizations the thread is not blocked until the notification is processed. Notifications perform a very similar function to that of synchronizations, but without the performance detriment.

Notificaitons do have some limitations however. One is that a value cannot be returned from the main thread to the calling thread, because notifications do not suspend the calling thread.

16.15 TIdThreadSafe

TIdThreadSafe is a base class for implementing threadsafe classes. TIdThreadSafe is never used itself and is only designed as a base class.

Indy contains the following ready to use descendants: TIdThreadSafeInteger, TIdThreadSafeCardinal, TIdThreadSafeString, TIdThreadSafeStringList, TIdThreadSafeList. These classes can be used to create threadsafe versions of integers, strings, etc. They can then freely be accessed from any number of threads and the details are taken care of automatically. In addition they support explicit locking for extended use.

16.16 Common Problems

The single biggest issue with threads is concurrency. Since threads execute in parallel, there is a concurrency issue with accessing common data. When using threads in an application, the following issues are commonly encountered:
Executing Indy clients in threads brings up the following concurrency issues:

1. Updating the user interface from a thread.
2. Communicating to the main thread from secondary threads.
3. Accessing data in the main thread from a secondary thread.
4. Returning result data from a thread.
5. Detecting when a thread has finished.

16.17 Bottlenecks

Many developers create multi threaded applications that function well when the number of concurrent threads is small, however the applications quickly bog down when the number of threads is increased. This is usually due to the bottleneck effect. The bottleneck effect is when a certain piece of code blocks other threads from executing and other threads must wait on that piece of code to complete. No matter how fast the other code is, the slowest point is the bottleneck. That is, code can only be as fast as the slowest part.

Many developers instead of looking for bottlenecks spend their time improving parts of code that they suspect is not "fast enough" but when compared to the existing bottlenecks has negligible if any effect.

Usually eliminating one bottleneck will gain a net increase in speed faster than dozens if not hundreds of other optimizations. Because of this, focus on the bottlenecks first. Then and only then, should other code be looked at for possible optimization.

Some common bottlenecks to avoid are covered next.

16.17.1 Critical Section Implementation

Critical sections are an effective and lightweight manner to control access to a resource so that only one thread can access a given resource at a time.

Often one critical section is used to protect multiple resources. That is given resources A, B and C, one critical section is used to protect them collectively, yet each resource is independent. The problem arises in that while B is in use, A and C are locked as well. Critical sections are rather lightweight and a dedicated critical section should be used for each resource.

Critical sections also sometimes lock too much code. The amount of code between the Enter and Leave methods of a critical section should be kept to an absolute minimum and in most cases multiple critical sections used instead if possible.

16.17.2 TMREWS

The TMREWS can provide a significant performance boost over critical sections if most access is limited to reading and only some access requires writing. However, the TMREWS class is heavier weight than a critical section and takes more code to obtain a lock. For very small pieces of code, even if writing is minimal, usually a critical section will perform better than a TMREWS.

16.17.3 Syncrhonizations

Synchronizations are usually used to perform user interface updates.

The problem with synchronization is that the calling thread is paused until the synchronization is complete. Since the main thread processes all synchronizations, only one can be processed at a time and thus they are serialized. This causes all synchronizations globally to become one large bottleneck.

Whenever possible, notifications should be used instead.

16.17.4 User Interface Updates

Multi threaded applications often perform far too many user interface updates. Much of the performance gains achieved by multi threading an application can quickly be lost by delays caused by user interface updates. In many cases this will cause a multi threaded application to execute slower than its single threaded version.

Special consideration should be taken with server implementations. A server is a server and its main responsibility is to its clients. User interface is secondary to its function. Thus it is perfectly acceptable to make the user interface a separate application which itself is a special client to the server. Another option is to use techniques such as notifications or batched updates. Both of these cause your user interface to be slightly delayed, but its is much better to have a user interface that is one second delayed than for 200 clients to be each delayed 1 second. With servers, clients are king.

Section XVI

Servers

17 Servers

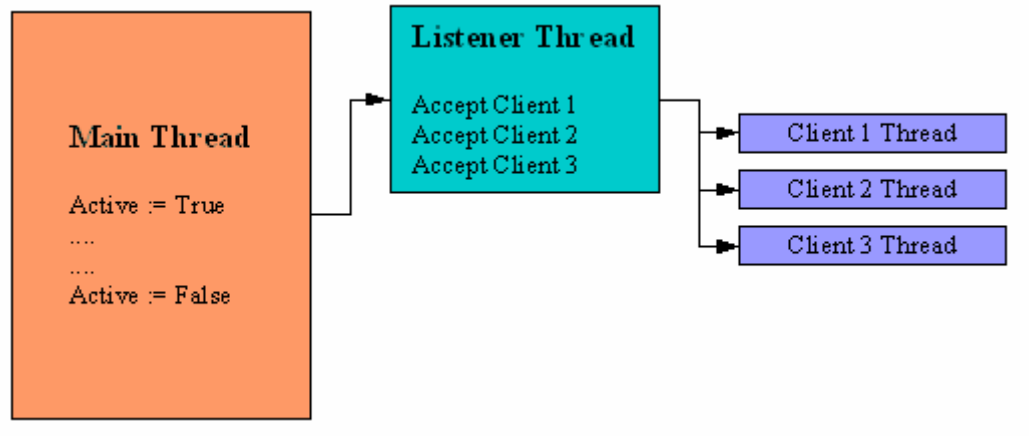
Indy has a variety of server models depending on your needs and the protocol used. The next few sections will provide an overview the base Indy server components.

17.1 Server Types

17.1.1 TIdTCPServer

The most prominent Indy server is TIdTCPServer.

TIdTCPServer creates a secondary listening thread that is independent from the main thread of the program. The listener thread listens for incoming requests from clients. For each client that it answers, it creates a new thread to specifically service that individual client connection. The appropriate events are then fired within the context of that thread.



17.1.1.1 The Role of Threads

Indy servers are designed around threads and operate in a manner similar to how Unix servers operate. Unix applications typically interface to the stack directly with little or no abstraction layer. Indy isolates the programmer from the stack using a high level of abstraction and internally implements many details that can be automatic and transparent.

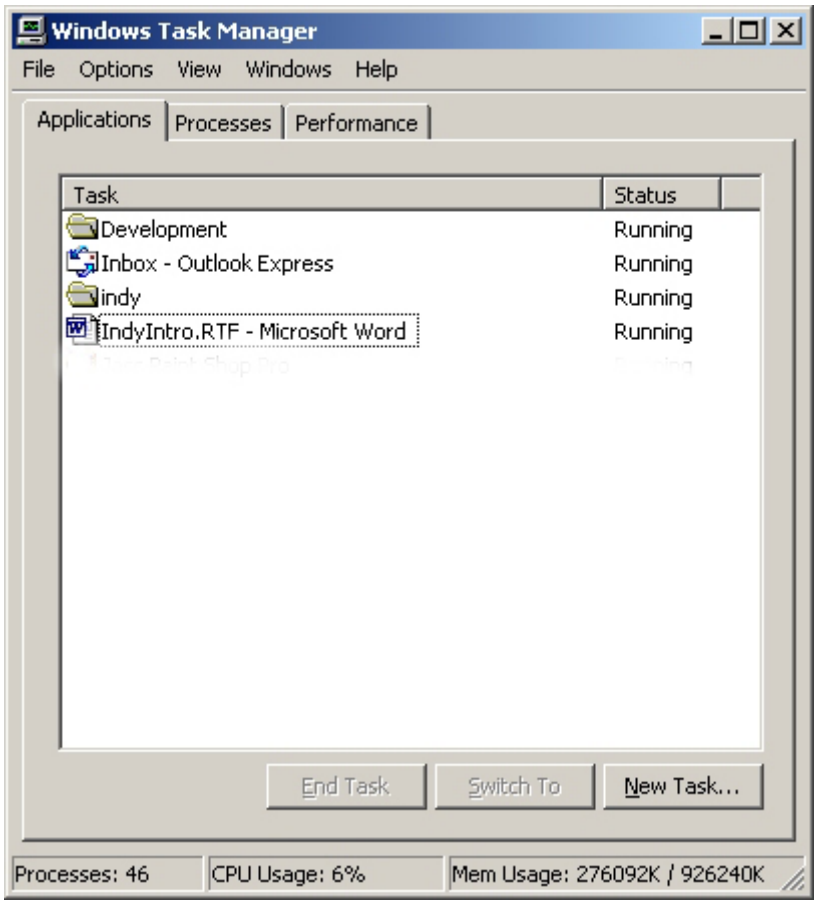
Typically, Unix servers have one or more listener processes that watch for incoming requests from clients. For each client request that the listener process accepts, the server forks a new process to handle each client connection. Handling multiple client connections in this manner is very easy as each process deals with only one client. The process also runs in its own security context, which can be set by the listener or the process itself, based on credentials, authentication, or other means.

Indy servers operate in a very similar fashion. Windows unlike Unix does not fork well, but Windows does perform threading quite well. Indy servers allocate a thread for each client connection instead of a complete process as Unix does. This provides for nearly all the advantages of processes, with none of the disadvantages

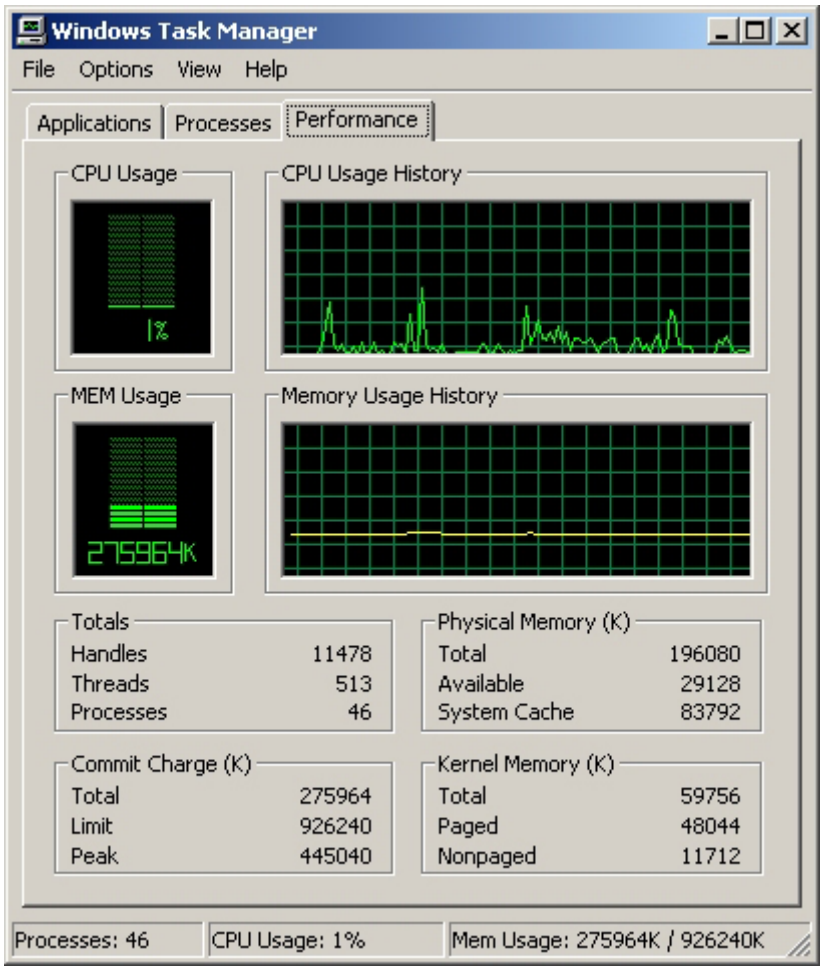
17.1.1.2 Hundreds of Threads

With a busy server, hundreds or even thousands of threads can easily be needed. There is a common misconception that hundreds of threads will kill your system. This is a false belief.

The number of threads that your system has running now may surprise you. With only minimal services started and the following applications running:



Here is a screenshot from my system which has 333 threads currently created:



Even with 513 threads, you can see that CPU utilization is only at 1%. A heavily used IIS (Microsoft Internet Information Server) will create hundreds or thousands more threads. This test was performed on a Pentium III 750 MHz with 256 MB of RAM.

With most servers, threads spend most of their time waiting on data. While waiting on blocking calls, the thread will be inactive. Thus in a server with 500 threads only 25 may be active at a single time.

In socket applications there are further restrictions dictated by the slowest component. In network applications the network card is typically one of the slowest components and thus nothing can exceed the capacity of the network card. Even a fast network card is many factors slower than a CPU and in most cases will be the bottleneck even if fully utilized.

17.1.1.3 Realistic Thread Limits

Realistically the average system will start to see problems with a process when the process creates more than 1,000 threads because of memory issues. The stack size of the threads can be reduced to increase the number of threads, however at this point other alternatives should be investigated.

Most servers only require a few hundred threads at most. However very high volume servers, or servers which have many low traffic but connected clients such as a high volume Chat server will need to be implemented in a different fashion. Such servers using Indy would create several thousand threads.

It is also important to understand that the number of clients is not necessarily the same as concurrent threads. While each client is allocated an individual thread, a thread is only allocated while the client is connected. Many servers service short lived connections such as an HTTP server. HTTP connections for web pages typically only last a second or less, especially if proxying or caching is used. Assuming 1 second per connection and only 500 threads, this would allow for 30,000 clients per hour.

Indy 10 addresses these needs by allowing other models in addition to threading of servers. Indy 10 is limited only by available memory to allocate sockets.

17.1.1.4 Server Models

There are two ways to build TCP servers, with command handlers and the OnExecute event. Command handlers make building servers much easier, but do not suite all situations.

Command handlers are suited to protocol which exchange command in text format, but are not suited to protocols which have a binary command structure or no command structure. Most protocols are text based and command handlers can be used. Command handlers are completely optional. If not used, Indy servers still support the old methods of use. Command handlers are covered in more detail in the command handlers section .

Some protocols are binary or have no command structure and are not suited for command handlers. For such servers the OnExecute event should be used. OnExecute occurs repeatedly as long as the connection is alive and passes the connection as its argument. A very simple server implemented using OnExecute might look like this:

```
procedure TformMain.IdTCPServer1Execute(AThread: TIdPeerThread);
var
  LCmd: string;
begin
  with AThread.Connection do begin
    LCmd := Trim(ReadLn);
    if SameText(LCmd, 'QUIT') then begin
      WriteLn('200 Good bye');
      Disconnect;
    end else if SameText(LCmd, 'DATE') then begin
      WriteLn('200 ' + DateToStr(Date));
    end else begin
      WriteLn('400 Unknown command');
    end;
  end;
end;
```

There is no need to check for a valid connection as Indy does that automatically. There also is no need to perform any looping as Indy will also do that for you. It call the event repeatedly until there is no longer a connection. This can be caused either by an explicit disconnect a shown above, by a network error, or by the client disconnecting. In fact, no looping should be done as it might interfere with Indy's disconnect detection. If looping must be done internally to this event special care must be take to allow such exceptions to bubble up to Indy so that it can handle them as well.

17.1.1.5 Command Handlers

Indy 9.0 contains a new feature called command handlers. Command handlers are a new concept used in TIdTCPServer that allow the server to do command parsing and processing for you. Command handlers are a sort of "visual server management" and are just a very small sneak peak into the future of Indy servers.

For each command that you want the server to handle, a command handler is created. Think of command handlers as action lists for servers. The command handler contains properties that tell it how to parse the command including how to parse parameters, the command itself, some actions that it can possibly perform itself, and optional auto replies. In some cases using the properties alone, you can create a fully functional command without having to write any code. Each command handler has a unique OnCommand event. When the event is called there is no need to determine which command has been requested as each event is unique to a command handler. In addition, the command handler has taken optional actions for you already, and parsed the parameters for your use.

Here is a very basic demo of how to use command handlers. First we must define the two commands: QUIT and DATE. To do this two command handlers are created at design time as shown here:

For cmdhQuit the disconnect property is set to true. For cmdhDate the OnCommand event is defined as shown here:

```
procedure TForm1.IdTCPServer1cmdhDateCommand(ASender:  TIdCommand);
begin
  ASender.Reply.Text.Text := DateTimeToStr(Date);
end;
```

This is the complete code for the command handler version. All other details have been specified by setting properties of the command handlers.

Command handlers are covered in more detail in the command handlers section.

17.1.2 TIdUDPServer

Since UDP is connectionless (define), TIdUDPServer operates differently than TIdTCPServer. TIdUDPServer does not have any modes similar to TIdSimpleServer, but since UDP is connectionless, TIdUDPClient does have single use listening methods.

TIdUDPServer when active creates a listening thread to listen for inbound UDP packets. For each UDP packet received, TIdUDPServer will fire the OnUDPRead event in the main thread, or in the context of the listening thread depending on the value of the ThreadedEvent property.

When ThreadedEvent is false, the OnUDPRead event will be fired in the context of the main program thread. When ThreadedEvent is true, the OnUDPRead event is fired in the context of the listener thread.

When ThreadedEvent is true or false, its execution will block the receiving of more messages. Because of this the processing of the OnUDPRead event must be quick.

17.1.3 TIdSimpleServer

TIdSimpleServer is for creating single use servers. TIdSimpleServer is intended to service a single connection at a time. While it can service another request after it is finished with a request, it is typically used for single use requests.

TIdSimpleServer will not spawn any listener or secondary connection threads. All of its functionality occurs from within the thread that it is used.

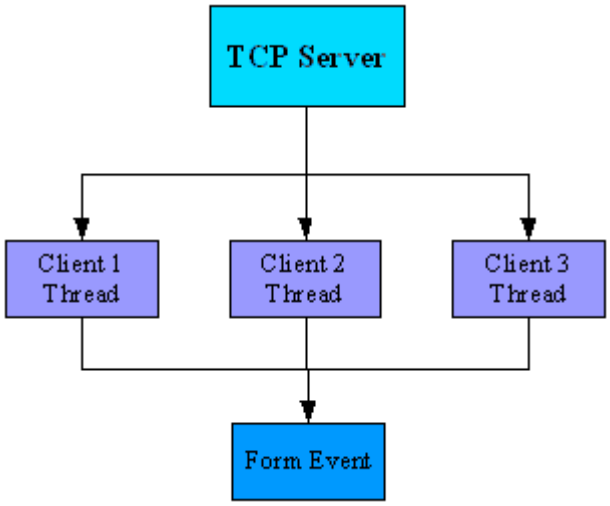
TIdFTP client component utilizes TIdSimpleServer. When FTP performs a file transfer, a secondary TCP connection is opened to transfer the data, and closed when the data has been transferred. This connection is called the "data channel" and is unique for each file transfer.

17.2 Threaded Events

TIdTCPServer events are threaded. This means that while they are not part of a thread class, they are executed from with in a thread. This is a very important detail. Please be sure to understand this detail before proceeding.

This might seem a bit confusing at first how the event can appear as part of the form, yet execute from within a thread. However it was constructed this way intentionally so that events could be created at design time just like any other event, without the need to create custom classes and provide overrides.

If descendant components are being built, overrides are available as well. However for building applications, the event model is much easier to use.



Each client is assigned its own thread. Using that thread the events of the TCP server (which are parts of the form or data module when created) are called from those threads. This means that a single event may be called multiple times from several threads. Such events receive an AThread argument which specifies the thread that is calling the event.

Examples of threaded events are the server OnConnect, OnExecute, and OnDisconnect.

17.3 TCP Server Models

Indy's TCP server supports two models for constructing servers. These methods are OnExecute, and command handlers. Indy 8 only supported OnExecute.

17.3.1 OnExecute

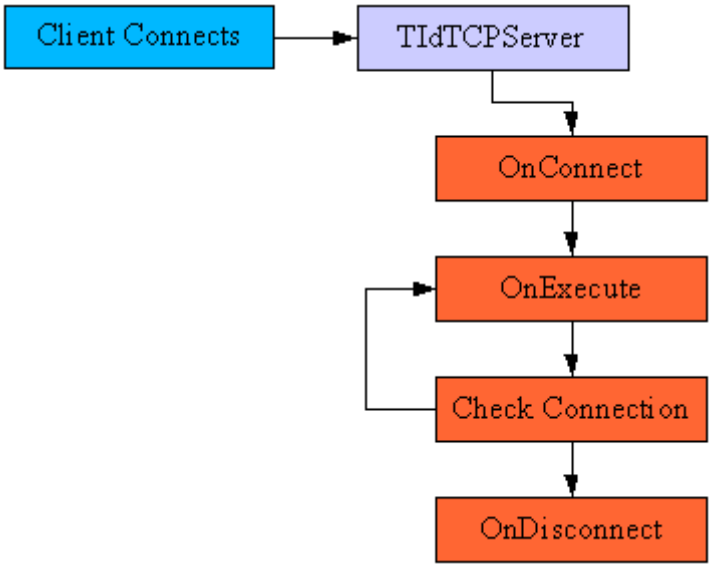
OnExecute refers to the OnExecute event of TIdTCPServer. When implementing a server using this model, the OnExecute must be defined, or the DoExecute method must be overridden.

Using the OnExecute model allows complete control by the developer and allows implementation of any type of protocol including binary protocols.

After a client connects to a server the OnExecute will be fired. If no OnExecute is defined, an exception will be raised. The OnExecute event if fired from inside of a loop as long as the client is connected. This is a very important detail, and because of this developers must be cautious to:

1. Remeber the event is inside a loop.
2. Not implement looping that will interfere with Indy.

The loop is constructed internally in Indy as shown in this diagram:



The Check Connection step performs the following checks:

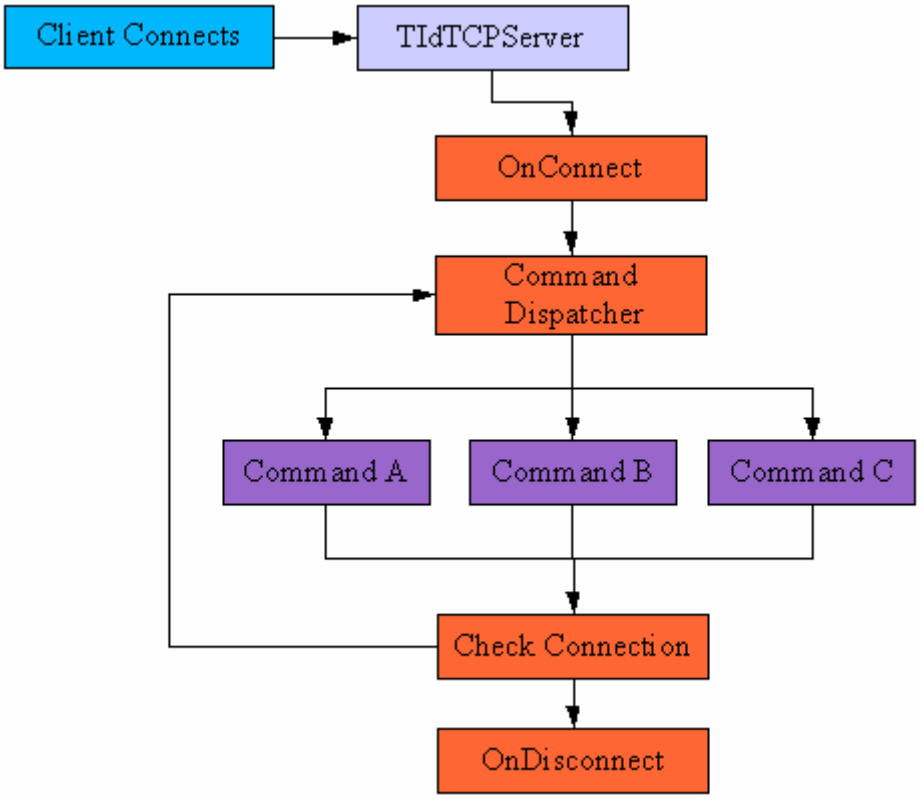
- Client is still connected
- Disconnect has not been called during OnExecute
- There were no fatal errors
- There were no unhandled exceptions in OnExecute
- Server is still active

If all of these conditions and other checks are true, OnExecute is called again. Because of this, developers should never construct their own looping code inside OnExecute which attempts to replicate this as it will interfere with Indy.

17.3.2 Command Handlers

Command handlers are like action lists for building servers in a RAD fashion using a design time environment. Command handlers are limited to conversational text protocols. The data portions of the protocols can however contain binary data.

Command handlers automatically read and parse the command. The specific OnCommand event is then fired.



17.4 Command Handlers

Creating servers in Indy has always been fairly straightforward, however with Indy 9 it has become even easier with the introduction of command handlers to Indy's TCP server (TIdTCPServer).

Command handlers are in a fashion, action lists for servers. Command handlers work in such a fashion that you define a command handler for each command, and then using that command handler define the behavior and responses for that particular command. When a command is received from the client, the server automatically parses it and passes it to the appropriate command handler. Command handlers not only have properties to customize their behavior, but have methods and events as well.

Command handlers only work with text based command and response (conversational) TCP protocols. However this covers about 95% of the servers that are in common use today. While command handlers can deal with binary data, they can only deal with text commands. There are some protocols which use binary commands. For protocols using binary commands, or text commands that are not conversational, the implementation of Command Handlers also preserves backwards compatibility and allows servers to be implemented without them.

17.4.1 Implementation

TCPServer contains a property named CommandHandlers which is a collection of command handlers. Command handlers are usually created at design time, however for protocol implementing descendants they can also be created at run time. If command handlers are created at run time they should be created by overriding the InitializeCommandHandlers method. This will ensure that they are only created a run time. If they are created in the constructor they will be created each time the TCPServer is streamed in, and when it is streamed out they will be saved. Soon there will be many

copies of each command handler. Initialize is called after the TCPServer is activated for the first time.

TCPServer contains several properties and events relating to command handlers. CommandHandlersEnabled enables or disables command handler processing as a whole. OnAfterCommandHandler is fired after each command handler is processed and OnBeforeCommand handler is fired before each command handler is processed. OnNoCommandHandler is fired if no command handler is found that matches the command.

If CommandHandlersEnabled is true and command handlers exists, command handler processing is performed. Otherwise the OnExecute event is called if assignd. OnExecute is not called if command handler processing is performed.

As long as the connection exists, TCPServer will read lines of text from the connection and attempt to match command handlers to the data. Any blank lines will be ignored. For non blank lines first OnBeforeCommandHandler will be fired. Next a matching command handler will be searched for. If a matching command handler is found and its enabled property is true, its OnCommand event will be fired. Otherwise OnNoCommandHandler is fired. Finally, OnAfterCommand handler will be fired.

17.4.2 Example Protocol

To demonstrate a basic implementation of command handlers a simple protocol will be defined. For the sake of demonstration a custom time server will be implemented consisting of three commands:

- **Help** - Display a list of supported commands and basic help on each.
- **DateTime <format>** - Return the current date and/or time using the specified format. If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
- **Quit** - Terminate the session and disconnect.

This is a very basic implementation, but will work quite nicely for demonstration purposes. You may wish to expand on it to further play with the capabilities of command handlers.

17.4.3 Base Demo

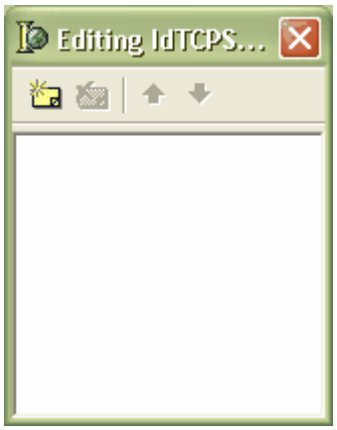
First lets construct the base of the demo which will then be built upon. It is assumed that you are already familiar with TIdTCPServer and what the following steps do. To build the base of the demo perform the following steps:

1. Create a new application.
2. Add a TIdTCPServer to the form.
3. Set TIdTCPServer.Default to 6000. 6000 is just an arbitrary number and any free port can be used.
4. Set TIdTCPServer.Active to True. This will activate the server when the application is run.

This will create the base application. It does not do anything yet though as no events or command handlers have been created.

17.4.4 Creating a Command Handler

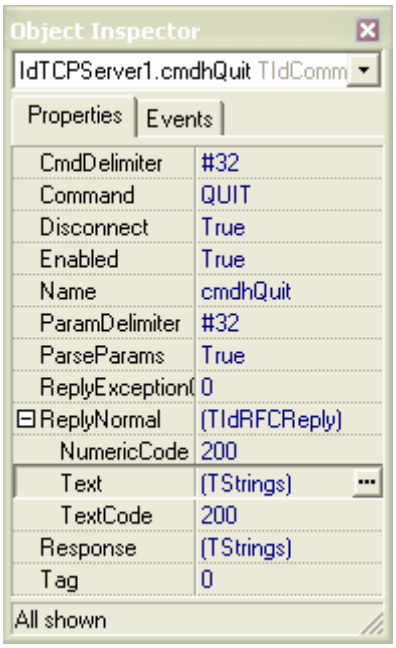
Command handlers are defined by editing the CommandHandlers property of TIdTCPServer. The CommandHandlers property is a collection. Command handlers can be modified at run time and/or design time. To edit command handlers at design time select the ellipsis button of the CommandHandlers property in the Object Inspector. A dialog like this should appear:



It is empty because no command handlers have been created. To create a command handler either right click on the white area and select add, or select the first button on the toolbar. After doing this a command handler will be listed in the property editor as shown here:

To edit the command handler this property editor is used to select the command handler, and then the object inspector is used to edit its properties and events. Editing command handlers is the same as editing fields of a dataset or columns of a DBGrid. If the object inspector is not visible, press the F11 key to display it.

The object inspector will appear similar to this one. This one has some properties modified already from the defaults for implementing the QUIT command and will be covered next.



Step by step here are the properties that have been modified to implement the QUIT command:

1. Command = Quit - This is the command that the server use to match input from the client and determine which command handler will handle the command. Command is case insensitive.
2. Disconnect = True - This tells the server to disconnect the client after this command has been processed.
3. Name = cmdhQuit - This has no effect on the behavior of the command handler, but makes it easier to identify in code if necessary. This step is optional.
4. ReplyNormal.NumericCode = 200 - Commands are typically replied to with a 3 digit numeric code and optional text. This tells the command handler to reply with 200 plus any text found in ReplyNormal.Text if no errors occur during processing of the command.
5. ReplyNormal.Text = Good Bye - This text is also sent with ReplyNormal.NumericCode.

A fully functional command handler has now been created.

17.4.5 Command Handler Support

An initial command handler has been created, however there are several global options relating to text based servers and command handlers that should be set as well. All of the properties discussed here are properties of the TIdTCPServer itself and not individual command handlers.

Greeting

It is common practice for servers to provide a response upon connection before the server receives commands from the client. A typical reply that indicates the server is ready is 200, and setting it to non zero will enable sending of the greeting.

Set Greeting.NumericCode = 200 and Greeting.Text to "Hello".

ReplyExceptionCode

If any unhandled exceptions occur during the processing of a command this number will be used to construct a reply to the client if its value is non zero. 500 is a typical reply for internal unknown errors. For the text portion of the reply the exception text will be sent.

Set ReplyExceptionCode to 500.

ReplyUnknown

If no matching command handler is found for a command issued by a client, TIdTCPServer's ReplyUnknownCommand property is used to return an error to the client. 400 is a common reply for internal errors.

Set ReplyUnknown.NumericCode to 400 and ReplyUnknown.Text to "Unknown Command".

Other Properties

There are other properties and even events of TIdTCPServer for implementing additional behaviors related to command handlers, but the ones listed here are the ones that should be implemented as a minimum.

17.4.6 Testing the New Command

Now that an initial command has been created it can be tested easily using telnet since command handlers are text based. To test the new command:

1. Run the application.
2. From the Start : Run dialog enter: *telnet 127.0.0.1 6000* and press OK. This instructs telnet to connect to the computer on port 6000 which is the port that the demo is listening on.
3. The server should reply with *200 Hello* which is the greeting that was defined in the Greeting property of TIdTCPServer earlier.
4. Telnet will then display a caret. This means that the server is ready and waiting for input (i.e. a command).
5. Type *HELP* and press enter. The server responds with "400 Unknown Command". This is because no HELP command has been implemented yet, and the "400 Unknown Command" was what was defined in the ReplyUnknown property.
6. Type *QUIT*. The server responds with "200 Good Bye" and disconnects the client.

Congratulations! You have just built a server using command handlers. The next section will progress with implementing the other two command HELP and DATETIME which have different behaviors and needs from QUIT.

17.4.7 Implementing HELP

The HELP command is similar in behavior to the QUIT command with these two differences.

1. It does not disconnect.
2. In addition to the status reply, it also provides a textual response with the help information.

To implement the HELP command perform the following steps:

1. Create a new command handler.
2. Command = Help
4. Name = cmdhHelp
5. ReplyNormal.NumericCode = 200
6. ReplyNormal.Text = Help Follows

All of these steps you should be familiar with as they are similar to those implemented in QUIT. The additional property that is used for implementing the textual form of the response is the Response property which is a string list. If Response contains text, it will be sent to the client after ReplyNormal is sent. For implementation of the HELP command use the string list property editor for the Response property and enter:

```
  Help - Display a list of supported commands and basic help on each.
  DateTime <format> - Return the current date and/or time using the specified
      format.
    If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
  Quit - Terminate the session and disconnect.
```

Now if you connect to the server and send the HELP command the server will reply as follows:

```
  200 Hello
  help
  200 Help Follows
  Help - Display a list of supported commands and basic help on each.
  DateTime <format> - Return the current date and/or time using the specified
  format.
    If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
  Quit - Terminate the session and disconnect.
.
```

17.4.8 Implementing DATETIME

DATETIME is the final command in the implementation of this protocol. It differs from either QUIT or HELP in that it requires some custom functionality that cannot be created merely by using properties. In the implementation of DATETIME an event will be used to implement this custom behavior.

First build the base command handler using steps you are already familiar with:

1. Create a new command handler.
2. Command = DateTime
3. Name = cmdhDateTime
4. ReplyNormal.NumericCode = 200

This time a ReplyNormal.Text was not defined, the event will custom defined it for each request. To define the event, use the Object Inspector while the DATETIME command handler is selected. Switch to the events tab and create an OnCommand event. Delphi will create a event shell as shown next:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
begin

end;
```

OnCommand passes in an argument of ASender which is of type TIdCommand. This is not the command handler, but the command itself. Command Handlers are global to all connections, while commands are specific to the connection being handled by this instance of the OnCommand event. This ensures that the event can provide specific behavior to each client connection.

Before the event is called, Indy will create an instance of the command and initialize its properties based on the command handler. You can then use the command to change properties from their defaults, call methods to instruct the command to perform tasks, or access its Connection property to interact with the connection directly.

This protocol defines DATETIME as accepting an optional parameter specifying the format of the date and time to be returned. The command has support for this as well in the Params property, which is a string list. When a command is received from the client, if the command handler's ParseParams property is True (True is the default) Indy will use the CmdDelimeter property (which defaults to #32 or space) to parse the command into the command and its parameters.

For example in this protocol the client may send:

```
DATETIME hhnnss
```

In this case, ASender.Params would contain the string "hhnnss" in ASender.Params[0]. The number of parameters can be determined by reading ASender.Params.Count.

Using these properties the OnCommand can be implemented as follows:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end else begin
    LFormat := ASender.Params[0];
  end;
  ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

This implementation merely reads the parameters and uses ASender.Reply.Text to send the reply back to the client. It is not needed to set ASender.Reply.NumericCode as Indy initializes it to 200 from the command handler's ReplyNormal.NumericCode.

Note the use of ASender.Reply.Text.Text. Text is required twice because the Text property of the command is a string list, and we are accessing TStrings.Text in addition to that. Since it is a string list, other methods or properties such as Add, Delete, etc may also be used. Text is used here as ASender.Reply.Text may be pre-initialized in some cases and using ASender.Reply.Text.Text will overwrite any preexisting text.

If the demo is tested again using telnet, it will yield results similar to the following:

```
200 Hello
datetime
200 2002-08-26 18:48:06
```

In some cases Params cannot be used. DATETIME is one of them. Consider if the user sends this as a command:

```
DATETIME mm dd yy
```

In this case Params.Count would be 3, and the event would fail and return only the value for months (mm). For cases where the parameter has embedded delimiters the UnparsedParams property of the command should be used instead. Optionally the ParseParams property can be set to False. UnparsedParams will contain the data irregardless of the value of ParseParams, but setting it to false will increase efficiency by telling Indy that there is no need to parse the parameters into the Params property.

The event with the code modified to use UnparsedParams follows:

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end else begin
    LFormat := ASender.UnparsedParams;
  end;
  ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

17.4.9 Conclusion

Command handlers are very flexible and contain many more properties and methods than covered here. This is only an introduction to command handlers and their capabilities. Hopefully enough has been covered to pique your interest and get you started.

There are also special plans for future versions of Indy to make command handlers even more visual in the design phase.

17.5 Postal Code Server - OnExecute Implementation

OnExecute Implementation
Demo
Threads

17.6 Postal Code Server - Command Handlers

Command Handlers
Greeting
CommandHandlers
ReplyException
ReplyTexts
ReplyUnknownCommand
Demo

17.7 Thread Management

Thread management is abstracted in Indy into thread managers. Thread managers allow for different or even custom implementations of thread management.

The use of thread managers is optional. If you do not specify a thread manager to use by setting the ThreadManager property of a component that supports thread management (such as TIdTCPServer) Indy will implicitly create and destroy an instance of the default thread manager, TIdThreadMgrDefault .

17.7.1 TIdThreadMgrDefault

Default thread management in Indy is very simple and basic. Each time a thread is needed, one is created. When it is no longer needed, it is destroyed. For most servers this is acceptable and unless you have justified the need of a thread pool you should use the default thread management. In most servers the difference in performance will be negligible or none at all.

It also has the added advantage that every thread is guaranteed to be "clean". Threads often allocate memory or other objects. These objects are usualy cleaned up automatically when the thread is destroyed. Thus using default thread management will help to ensure that all memory is freed as well as cleared. When thread pooling is used if you store data associated with the thread you must be sure to clean it before the thread is reused again. Failure to do so could provide users with sensitive information from a previous user. No such precautions are required with the default manager since the thread and associated data is destroyed each time.

17.7.2 Thread Pooling

Normally default thread management is suitable. However, for servers which service short lived connections the creation and destruction of threads to service each request consumes a considerable amount of time compared to servicing the request. In a situation such as this thread pooling should be used.

With thread pooling threads are recycled and also preallocated. In a thread pool threads are created before they are needed and kept inactive in a pool. When a thread is needed, one is taken from the pool and activated. If more threads are needed than available from the pool more threads are created. When a thread is no longer required, instead of being destroyed it is deactivated and redeposited into the pool so it can be reused later when needed.

The creation and destruction of threads can be resource intensive. This is especially evident with servers that have short-lived connections. Such servers create a thread use it for very brief time and then destroy it. This causes for a very high frequency of creation and destruction of threads. An example of this is a time, or even and web server. A single request is sent, and a simple answer returned. When using a browser to browse a web site hundreds of connections and disconnections to the server may occur.

Thread pooling can alleviate such situations. Instead of creating and destroying threads on demand, threads are borrowed from a list of inactive but already created list (pool). When a thread is no longer needed, it is deposited back into the pool instead of being destroyed. While threads are in the pool they are marked inactive and thus do not consume CPU cycles. For a further improvement, the size of the pool can be adjusted dynamically to meet the current needs of the system.

Indy does support thread pooling. Thread pooling in Indy can be achieved via use of the TIdThreadMgrPool component.

Section XVII

SSL - Secure Sockets

18 SSL - Secure Sockets

SSL stands for Secure Socket Layer and is the accepted encryption method of encrypting data on the Internet. SSL is mostly commonly used with HTTP (Web) traffic and referred to as secure HTTP (HTTPS). SSL however is not limited to HTTP and can be used with any of the TCP/IP protocols.

In Indy to use SSL, you must first install the proper SSL DLLs. Indy's SSL support is done in an open fashion, but the only currently implemented SSL library is OpenSSL. The OpenSSL library is available as a set of DLLs and is available for download separately from the main Indy distribution.

The US and other governments in their infinite wisdom and understanding of technology banned the export of certain encryption methods such as SSL. Because of this SSL technology cannot be placed on a website unless certain measures are taken to verify the location of each client wishing to download the technology. This is not only difficult to implement, but opens the website owners to a lot of legal liability.

The restriction only applied to electronic distributions and did not apply to all forms of printed source code. The restriction also applied only to exportation, not importation.

So for a brief while programmers printed the source code upon such things as T-Shirts, crossed borders, and retyped and compiled such code. After this occurred, countries which did not sign the treaty with the US were free to distribute such encryption in any form and since there was no restriction importation anyone could download the encryption technology in ready to use form.

Many of these issues have since then been solved and some governments have lightened up. However many export restrictions are still in place and vary from country to country. Because of this, all Indy SSL work is done in Slovenia, and all SSL encryption technology related to Indy is distributed from Slovenia. Slovenia has no restrictions on export of encryption.

In addition to exportation of encryption, some countries have restrictions on use or even possession of encryption technologies. You should check your countries laws before implementing or using SSL. Countries such as China, Iraq, and others have harsh and even death penalties for even possessing such technology.

18.1 Secure HTTP / HTTPS

Implementing HTTPS in Indy is very easy. Simply pass a secure URL instead of a standard URL, and the Indy HTTP client (TIdHTTP) will do everything automatically. To make a URL secure, simply change the http:// portion to https://.

Note that for a HTTPS session to be established, the web server that will be responding to the request must support SSL and have encryption certificates installed. The Indy HTTP client also does not automatically verify the server certificates, this is your responsibility.

18.2 Other Clients

SSL can easily be implemented with any Indy TCP client by the use of an SSL IOHandler. Normally encryption should be implemented using an Intercept instead of an IOHandler. SSL is implemented in Indy using an IOHandler instead of an Intercept because the SSL libraries actually perform the communications themselves. The data is returned and accepted in unencrypted form directly from the SSL libraries.

To make an Indy TCP client use SSL simply add a TIdSSLIOHandlerSocket to your form from the Indy Misc tab. Now set the IOHandler property of your TCP client to the TIdSSLIOHandlerSocket. To support basic SSL, this is all that is required. The TIdSSLIOHandlerSocket does have additional properties for specifying client side certificates and other more advanced SSL options.

18.3 Server SSL

Implementing SSL in a server is a little more complicated than implementing SSL in clients. With clients typically all that is needed is to hook to a TIdTCPClient or descendant to a TIdSSLIOHandlerSocket instance. This is because the server does the larger share of the SSL work.

To implement SSL in servers a TIdServerIOHandlerSSL is used. The TIdTCPServer's intercept property is used to hook it to a TIdServerIOHandlerSSL. However unlike a TIdSSLIOHandlerSocket (Client), a TIdServerIOHandlerSSL requires some additional steps. Specifically certificates must be installed. These certificates must be provided as files on disk and specified in CertFile, KeyFile, and RootCertFile, all of which are sub properties of the SSLOptions property.

Certificates typically are obtained from a trusted certificate authority. You can make your own certificates and be your own trusted authority, but none of the browsers will trust your certificates and the browsers will display a warning when connecting to your server. If you wish to deploy to the Internet you must obtain a certificate from a certificate authority that the standard browsers trust. Verisign is the only authority trusted by all browsers. Thawte can also be used, but not all browsers trust Thawte by default.

If your clients are within your control such as on a intranet or extranet, you may choose to be your own trusted authority. To avoid the warning dialog however, you must install your certificate into each browser that will connect to your server.. This allows the browser to consider the signatures on your certificates as being from a trusted authority.

Note that this only applies to HTTP servers, but SSL is not limited to HTTP. You can implement SSL if you are implementing the client and the server and have full control over trust associations and rules.

18.4 Converting Certificates to PEM Format

Chances are that your certificates were not delivered to you in .pem format. If they are not in .pem you must convert them for use with Indy.

This procedure assumes that you have already received your key and certificate pair from some Certificate Authority (like Verisign or Thawte) and that you have them installed in Microsoft Internet Explorer in Personal Certificates Store.

18.4.1 Exporting the Certificate

Select the certificate and export it as a .pfx file (Personal Exchange Format). You may optionally protect it with a password.

18.4.2 Convert .pfx to .pem

As part of the SSL download, a utility named openssl.exe was included. This utility will be used to convert your .pfx file.

To use openssl.exe, use the following format:

openssl.exe pkcs12 –in <your file>.pfx –out <your file>.pem

Openssl.exe will prompt you for a password. Enter it if you used one, or leave it blank if you did not specify one. It will also prompt you for a new password for the .pem file. This is optional, but if you protect it with a password be sure to create a OnGetPassword event in the SSL intercept.

18.4.3 Splitting the .pem File

If you examine the new .pem file with a notepad, you will notice that it consists of two parts. The two parts consist of the private key and the certificate (public key) part. There is also some addition information included. Indy requires that this information be separated into separate files.

18.4.4 Key.pem

Create key.pem with notepad and paste everything between and including these two statements:

-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----

18.4.5 Cert.pem

Create cert.pem with notepad and paste everything between and including these two statements:

-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----

18.4.6 Root.pem

The final file that Indy requires is the Certificate Authority certificate file. You can obtain this from the Internet Explorer in Trusted Root Certificate Authority dialog. Select the Authority that issued your certificate and export it in Base64 (cer) format. This format is also the same as PEM format so after export simply rename the file to root.pem

Section

Indy 10 Overview

19 Indy 10 Overview

Indy 10 is still under development. The consistency of the current code can be likened to that of dirty motor oil. In a few weeks it should be more solid and the consistency of a nice custard. Thus, any information covered here is subject to change prior to the final Indy 10 release. The information presented here is based on current code, design goals, and projected directions.

Indy 10 contains many new features, especially relating to the portion referred to as core. Indy 10 core has been further abstracted for easier expansion. Indy 10 core also contains many new capabilities, and large performance enhancements.

19.1 Changes

19.1.1 Separation of Packages

Indy 10 has been separated into two packages: Core and Protocols.

Core contains all of the core pieces, base client components, and base server components. Core does not implement any higher level protocols.

The protocols package uses core and implements higher level protocols such as POP3, SMTP and HTTP.

This allows the Indy Pit Crew to better focus on specific parts. It also benefits users who are implementing custom protocols and may not need the protocols package.

19.1.2 SSL Core

The SSL capabilities of Indy 10 are now completely pluggable. Prior to Indy 10, the SSL support was pluggable at the TCP level, however protocols such as HTTP which used SSL for HTTPS were fixed to use Indy's default SSL implementation of OpenSSL.

Indy 10 continues to include support for OpenSSL, however the SSL capabilities of Indy are completely pluggable at the core and protocol level for other implementations.

SSL and other encryption methods are in the works from both the third parties SecureBlackbox and StreamSec.

19.1.3 SSL Protocols

Indy 10 now supports both implicit TLS and explicit TLS in the following clients and servers:

- POP3
- SMTP
- FTP
- NNTP

The SASL support code has been redesigned so it is usable with POP3 and IMAP4. Indy 10 now supports anonymous SASL, plain SASL, OTP (one-time-only password system) SASL, external SASL, and Auth Login.

19.1.4 FTP Client

The FTP Client has been expanded in the following ways:

- MLST and MLSD are now supported. This provides a standard FTP directory list format that is easily parsed.
- A special combine command has been added for multipart upload support. Note that this does require a server that supports the COMB command such as GlobalScape Secure FTP Server or the server component in Indy 10.
- A XCRC command has been added for obtaining the CRC of files. Note that this does require a server that supports the COMB command such as GlobalScape Secure FTP Server or the server component in Indy 10.
- The client now supports the MDTM command for obtaining the last modified date
- OTP (One-Time-Only password) calculator built in and OTP now automatically detected
- FTPX or Site to site file fransfer (where a file is transfered between two servers) is now supported.

Note that FTPX transfers will only work if the server supports it (many administrators and developers now disable this capability for security reasons).
- FTP specific IP6 extensions have been added.

19.1.5 FTP Server

FTP Server now supports:

- MFMT command and MFF. http://www.trevezel.com/downloads/draft-somers-ftp-mfxx-00.html
- XCRC and COMB commands for supporting Cute FTP Pro's multi-part ifle update.
- Support for MD5 and MMD5 commands (http://ietfreport.isoc.org/ids/draft-twine-ftpmd5-00.txt)
- Support for some Unix switches that effect how the directory is given and this includes Recursive directory lists (-R).
- Easily Parsed List format is supproted on the FTP server. (http://cr.yp.to/ftp/list/eplf.html).
- OTP calculator user manager can be used with the FTP server.
- A virtual system component can now be used to make FTP Server much easier.
- FTP specific IP6 extensions have been added.

Also FTPX site to site transfers capability can now be disabled. This is done for security reasons as the Port and PASV commands are subject to abuses as described by:

- http://www.cert.org/tech_tips/ftp_port_attacks.html
- http://www.kb.cert.org/vuls/id/2558
- http://www.cert.org/advisories/CA-1997-27.html
- http://www.geocities.com/SiliconValley/1947/Ftpbounc.htm
- http://cr.yp.to/ftp/security.html

19.1.6 FTP List Parsing

Indy 10 contains a plug in FTP list parsing system with provided parsers for nearly every FTP server type in existence, and even a few which are probably no longer functioning.

If the chance arises that an unsupported system is encoutnered, a custom handler can be used.

FTP servers supported:

- Bull GCOS 7 or Bull DPS 7000
- Bull GCOS 8 or Bull DPS 9000/TA200
- Cisco IOS
- Distinct FTP Server
- EPLF (Easily Parsed List Format)
- HellSoft FTP Server for Novell Netware 3 and 4
- HP 3000 or MPE/iX including HP 3000 with Posix
- IBM AS/400, OS/400
- IBM MVS, OS/390, z/OS
- IBM OS/2
- IBM VM, z/VM
- IBM VSE
- KA9Q or NOS
- Microware OS-9
- Music (Multi-User System for Interactive Computing)
- NCSA FTP Server for MS-DOS (CUTCP)
- Novell Netware

- Novell Netware Print Services for UNIX
- TOPS20
- UniTree
- VMS or VMS (including Multinet, MadGoat, UCX)
- Wind River VxWorks
- WinQVT/Net 3.98.15
- Xecom MicroRTOS

19.1.7 Other

Other notable changes and improvements to Indy 10 include but are not limited to:

- Server intercepts have been added permitting you to log the FTP server and they work similarly to the client intercepts.
- Systat UDP and TCP client and servers have been added.
- A DNS server component has been added.
- HTTP connect through proxy support has been added.
- TIdIPAddrMon has been added for monitoring all IP addresses and all network adaptors
- IP6 support
- A One-Time-Only password system has been implmeneted as both an OTP calculator for the clients and as a user manager component. This supports MD4, MD5, and SHA1 hashes.

19.2 Core Rebuild

The core of Indy 10 has undergoing massive structural changes. This will break some core code of users, but efforts have been made to preserve as much protocol and application level compatibility as possible. At times it may that seem the Indy Pit Crew takes no notice about the impact of changes on end users. Howerver this is not true. Each interface change is evluated, and the benefits are weighed against the disadvatages. Changes that are made, are designed in a manner so as to permit easy upgrading of existing source code with minimal effort. The Indy Pit Crew uses Indy in both private and commercial offerings, so each change is felt by the team as well.

It is the belief of the Indy Team that progress is not without some sacrifice. Through small changes to interfaces, large gains can be achieved and this a better Indy. Without such changes, the future directions are hindered, and vestigal baggage is retained. Just as Winshoes to Indy 8, and then Indy 8 to Indy 9 increased abstraction, so will Indy 10.

One of Indy's primary tenets is that it is easy to program and that it is based on a blocking model. This allows for easy development and user code does not suffer from serialization. Indy's design goal is to be easy to use, and fast enough for 95% of the end user's needs.

However in very large installations this causes excessive context switching and accumulated high thread overhead. These limitations appear only in large installations, at around 2,000 concurrent threads in a well designed application. In most applications limitations in user code appear long before the limitations of Indy.

Traditionally to serve the remaining 5% of the users, easily written code must be abandoned for complex and hard to maintain code such as direct overlapped I/O, I/O completion ports, or highly divided work units using threads against non blocking sockets. Indy 10 preserves Indy's easy to use blocking model, while making performance increases internally. Indy 10 does this by using advanced network interfaces and efficiently translating such back to a user friendly blocking model. Indy 10 thus is intended to serve 99.9% of the community's needs, with only a very small minority still requiring custom code for very unusual situations.

Indy 10 achieves this in many ways, but fibers are the keystone to this achievement. A fiber is very similar to a thread, but it is more flexible, and has lower overhead than a thread if used properly.

19.2.1 IOHandler Restructuring

To provide performance enhancements the IOHandlers in Indy 10 have been restructured and given a more important role. Previously the role of an IOHandler was to do only very basic I/O consisting of only the following functions:

- Open (Connect)
- Close (Disconnect)
- Read raw data
- Write raw data
- Check connection status

This role allowed for alternate IOHandlers to be created that performed the I/O from sources other than a socket. Even the default socket I/O was implemented using a default IOHandler.

Because the functionality was minimal, implementing IOHandlers was very straightforward. However this also often caused the IOHandlers to perform their I/O with less efficient methods. For example, an I/O handler may be receiving data from a local file to write, but it would have no way of knowing such because it only has one write method for all data. Even if the I/O handler had the capability of a high performance file read, it could not use it.

Indy 10's IOHandlers implement not only the minimal low level methods, but high level methods as well. Such higher level methods were previously implemented by TIdTCPConnection.

IOHandlers can be created as before by implementing only the low level methods. The base IOHandler contains default implementations of the higher level methods which use the lower level methods. However each IOHandler can override additional higher level methods to provide optimized implementations specific to the IOHandler.

19.2.2 Network Interfaces

Indy 9 implemented only one network interface. On Windows this interface was Winsock and on Linux the stack. Indy 10 still implements these interfaces but also implements more efficient interfaces under Windows. At this time no other interfaces have been implemented under Linux, but may be in the future. The need is less urgent under Linux because of its networking semantics.

Some of the additional interfaces are not available under all versions of Windows and should be used only in server implementations, or systems with a very large number of client connections. There is no need for such implementations in normal client applications.

The additional network interfaces are:

- Overlapped I/O
- I/O Completion Ports

Normally to use overlapped I/O, and more specifically I/O completion ports, complicated and highly unintuitive code must be written. However with Indy 10, as before Indy takes care of all these details and presents the developer with a friendly easy to use interface.

19.2.3 Fibers

In addition to expanding our thread suport, Indy 10 contains support for fibers. What is a fiber? In short, its a "Thread" but one
that is controlled by code - not the operating system. In fact, a thread may be thought of as an advanced fiber. Fibers are similar to Unix user threads.

A thread is a basic unit that the operating system allocates time to. A thread contains its own stack, certain processor registers, and a thread context. Threads are automatically scheduled by the operating system.

In general, fibers do not provide advantages over a well-designed multi threaded application. However fibers combined with an intelligent scheduler that has pertinent information available to it significantly increase performance through large efficiency gains.

Multiple fibers can be run using a single thread. A single fiber can be run by multiple threads, although only one at a time. You can run multiple fibers inside. The code to perform this can be quite complex, but Indy handles all this for you. All Indy
components both clients and servers support fibers, and in a mostly transparent way.

By using fibers, Indy is also able to translate the complex and unfriendly lower layer network interfaces into a developer friendly interface.

Fibers have been implemented in Indy only in Windows at this time.

19.2.4 Schedulers

Indy's fiber weaver is a scheduler which schedulers fibers onto one or more threads. Fibers deposit work items into a work queue and then wait. When a fiber's work item has been completed, the scheduler puts the fiber in a list of fibers that can be scheduled.

Operating system schedulers schedule threads in an intelligent manner, however they have limited information about threads as each thread is common and generic among all tasks in a system. Operating system schedulers can only schedule based on the wait state, and priority of a thread.

Indy's fiber weaver (fiber scheduler) uses advanced information that is specific to the task to determine scheduling needs, priorities and wait states. By doing so Indy is able to drastically reduce the number of context switches that occur relative to the work completed. This results in significant performance gains.

A context switch is when one thread is suspended and another is scheduled. To perform this the operating system must preemptively interrupt the execution path of the processor and save the context of the thread by storing several processor registers in memory. It must then restore another thread by loading processor registers from another memory location, and resume the thread's execution path.

Thread schedulers must balance the need to reduce context switching, with the need to make sure each thread receives enough processor time. Switching to often increases overhead and cause overhead to increase beyond any benefits. Switching not often enough causes unnecessary inter thread wait dependencies, and sluggish responses because threads are not receiving enough processor time.

To manage this, operating systems define a quantum, or a maximum time slice that a thread may receive processor time per switch. In most cases a thread will yield prior to this time being reached by entering a wait state. Wait states occur explicitly, or more commonly implicitly by making an operating

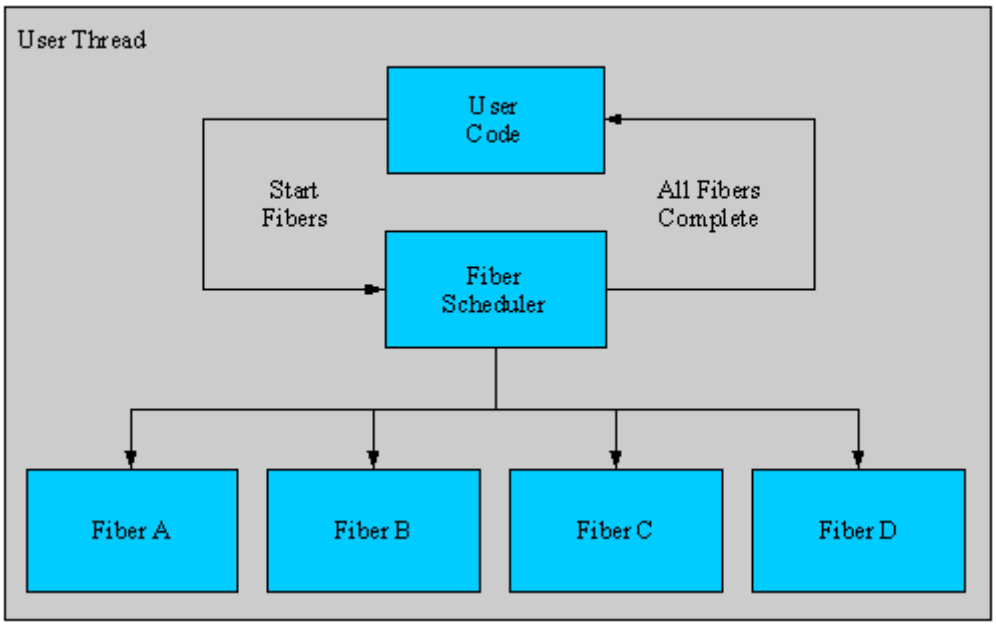
system call, or I/O call which cannot be completed immediately. When such a call occurs, the operating system accepts the yield and switches to another thread. If the thread is waiting on I/O or some other blocking operation it will be placed in a wait state and not considered for scheduling until the requested operation has completed, or timed out.

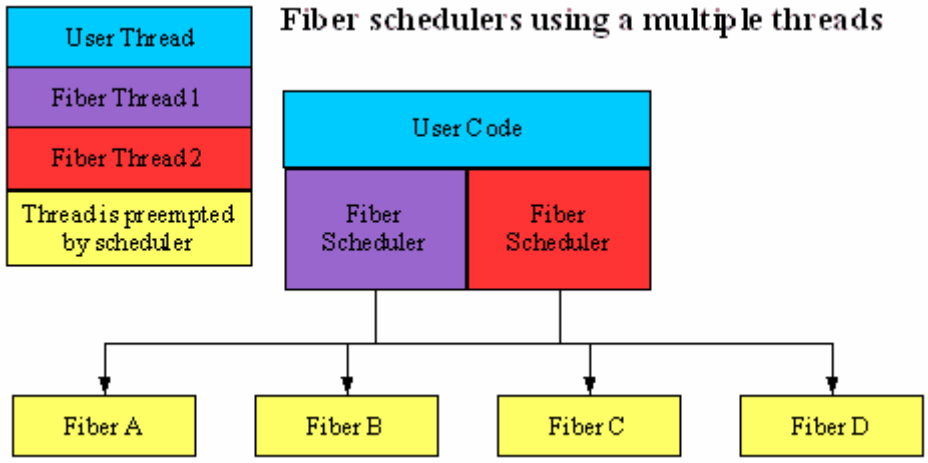
Indy's fiber weavers work in a similar fashion, but determine wait states at a much higher level using a wider range of information. Fibers can be determined before hand to be in wait states without the need to context switch to them and wait for them to enter a wait state. Indy also divides the work between fibers and chain engines which now perform much of the low level work.

The division of labor allows for more efficient network interfaces to be used such as I/O completion ports. I/O completion ports are more efficient because they run at a level closer to the hardware interface. Winsock and other calls which are farther from the hardware interface layer must communicate with the kernel to perform the actual call to the hardware interface. Calls which communicate with the kernel must undergo context switching to do so. Thus each Winsock call often involves many unnecessary context switches just to perform its function.

Schedulers can use a single thread, multiple threads, threads on demand, or even a thread pool.

Fiber scheduler using a single thread.





19.2.5 Work Queues

Work queues are first in first out (FIFO) queues which hold the work items requested by fibers. Most of this functionality is completely transparent to the average developer as Indy uses this functionality internally.

.

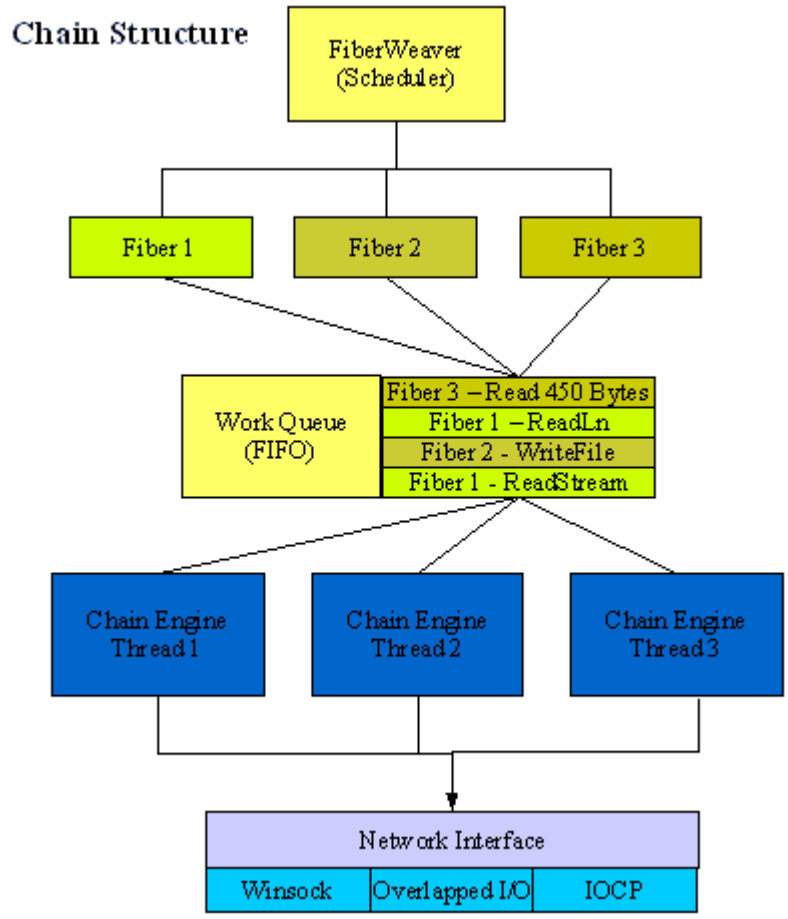
19.2.6 Chains

The system of work queues, schedulers, and chain engines in Indy are referred to as chains. While chains are used by Indy, they are not limited to Indy's internal use and have end user applications as well.

When chains are used, a chain based IOHandler deposits work items into the associated work queue. The fiber is then suspended until a unit of work for it has been completed. This is because the fiber can do nothing until the result of some
work is ready to be processed. Each IOHandler method is reduced to one more or more work tasks. For best performance, each method should resolve to as few specialized work tasks as possible.

A scheduler then manages the fibers while they wait for their work items to be processed.

Finally a chain engine processes the requests in the work queue and communicates with the schedulers.



19.2.7 Chain Engines

A chain engine is the lowest level of the chain system. The chain engine performs all of the actual input and output. The chain engine may consist of a single thread, or several.

The job of a chain engine is to extract tasks from a work queue and complete the tasks. Upon completion of each task, the chain engine notifies the scheduler and the scheduler evaluates which fiber should be considered for scheduling. The chain engine then moves on to the next task in the work queue.

If there are no items in the work queue, the chain engine remains idle.

Multiple chain engine types can be implemented to implement I/O completion ports, Winsock, Overlapped I/O, or other.

19.2.8 Contexts

In Indy 9 servers, connection specific data was stored as part of the thread class. This was achieved either by using the thread.data property or by descending from the peer thread class and adding new fields or properties for storage. This functioned because each connection has one thread that was specifically associated to that connection.

Indy 10 servers are not implemented in the same manner as Indy 9. Threads are not associated with a single connection. In fact it may not use threads at all, but instead fibers. Because of this, the previous method of storing data in the thread is no longer useful.

Indy 10 uses a new concept called contexts instead. Contexts hold connection specific data and are managed and tracked by Indy.

Section XX

Bonus Materials

20 Bonus Materials

This section contains bonus materials that are not directly related to Indy.

Enjoy!

20.1 Porting Delphi applications to Delphi.net

This article will cover the issues involved in porting existing Delphi code to DCCIL / Delphi.net. It will demonstrate things that are no longer permissible, as well as how to convert such parts to function properly in the .net framework.

This article will also provide guidance on estimating the work involved in porting an existing application to the .net framework using DCCIL / Delphi.net.

20.1.1 Terms

The focus of the article is on porting and effects on your Delphi code. Because of this, this article will not cover .net itself. It will only provide definitions for basic necessary terms relating to .net.

20.1.1.1 CIL

Common Intermediate Language, or just Intermediate Language for short. This is the language the compilers translate source code into for execution by the .net run time.

IL performs a similar role to P-Code or the Java interpreted language. It does however have its differences and is implemented differently.

20.1.1.2 CLR

CLR stands for Common Language Runtime. The CLR is the base of the .net framework and is the part that executes applications which are compiled to IL. At first glance the CLR may appear to be just a P-Code interpreter. While it performs a similar role it is not just a simple P-code interpreter, and does much more.

20.1.1.3 CTS

CTS stands for Common Type System. The CTS includes predefined "native" .net types that are accessible by any .net language. This means that integer is no longer defined by each compiler, but by the CTS and thus integer is exactly the same among all .net languages.

The CTS is not limited to just integer however and it includes many types. The CTS divides the types into two basic categories: value types, and reference types.

Value types are types that are stored on the stack. You might be familiar with the term simple or ordinal types. Value types include integers, bytes, other primitives, structures, and enumerations.

Reference types are types that are stored in the heap, but references are used to access them. Reference types include objects, interfaces, and pointers.

20.1.1.4 CLS

CLS stands for Common Language Specification. CLS is simply a specification that defines what features a language can, and must support to work with .net.

20.1.1.5 Managed Code

Managed code is code that is compiled to IL and executed by the CLR. The general goal of any .net application is to use 100% managed code.

20.1.1.6 Unmanaged Code

Unmanaged code is native compiled code such as that compiled by Delphi for Windows. Unmanaged code also includes native code DLL's, COM servers, and even the Win32 API. The goal of .net applications is to avoid unmanaged code as much as possible.

20.1.1.7 Assembly

An assembly is a collection of .net IL units. It is very similar to a Delphi package, and Delphi.net treats .net assemblies like Delphi packages.

20.1.2 Compilers / IDE's

There are several compilers and IDE's related to Delphi and .net.

20.1.2.1 DCCIL (Diesel)

DCCIL is the command line compiler for Delphi that produces .net output. DCCIL is what is commonly referred to as the "Delphi.net preview compiler" that is included with Delphi 7.

DCC is the name of the normal Delphi command line compiler and is short for exactly that, Delphi Command line Compiler.

IL is a reference to .net's Intermediate Language.

Thus DCC + IL = DCCIL. Saying D-C-C-I-L each time is cumbersome, so like other cumbersome counterparts it has been given a common name of "Diesel".

DCCIL has similar options to DCC with some specific extensions for .net.

20.1.2.1.1 Beta

DCCIL is currently beta and is not intended for production code. Because of this it does have bugs, and even many incomplete parts. This does not make it useless however, and Borland is regularly releasing updated versions.

Because it is beta, it also carries certain redistribution restrictions on programs created with DCCIL. Any output you create with DCCIL if redistributed must also be redistributed as beta.

20.1.2.2 Delphi 8

Borland has been pretty quiet about Delphi 8. However based on public statements we should expect to see a Delphi 8. However it will not be Delphi.net, but instead an extension of the Windows platform product.

20.1.2.3 SideWinder

There has been recent news of a SideWinder project from Borland. This is of course not its final name, but the code name Borland as assigned to it during development.

SideWinder is not Delphi.net either. SideWinder is a C# development environment for .net that will compete directly with Microsoft's Visual Studio.net.

20.1.2.4 Galileo

Galileo is Borland's code name for a reusable IDE framework. The first product to use this is SideWinder, and Galileo is expected to be the basis for the Delphi.net IDE when released as well.

20.1.3 Frameworks

20.1.3.1 .Net Framework

The .net framework is the class library that is the core of .net. This class library includes classes for I/O, data storage, simple types, complex types, database access, user interface and more. What the VCL is to Delphi programmers, is what the .net framework is to .net programmers.

The Win32 API is also gone, and has been replaced by classes in the .net framework which provide a better, and abstracted platform independent interface. Provisions exist to access the Win32 API directly, as well as true compiled DLL's. However using such access methods will make your code unmanaged and is not desirable in a .net application.

20.1.3.2 WinForms

WinForms is an assembly in the .net framework which includes classes for forms, buttons, edit boxes, and other GUI controls for building GUI applications. WinForms is a .Net managed interface to the Win32 API and is what Visual Studio.net uses for building GUI applications.

20.1.3.3 RTL

The RTL which consists of the non visual lower level classes in Delphi such as TList, TStrings, etc. The RTL is still available in Delphi.net. Much of what is in the RTL has similar counterparts in the .net framework, but by providing the RTL Borland has made it easier to port existing Delphi code without the need to rewrite large sections of code, and permits for cross platform code.

20.1.3.4 CLX

This is the part that gets confusing. Prior to Delphi 7 the following designations were used:

- VCL - Visual Component Library. The VCL collectively referred to the visual components, non visual components, and the RTL.
- CLX (Pronounced "Clicks") - Component Library for Cross Platform - CLX referred to the new cross platform version of the visual part of the VCL that was based on QT and ran on Linux as well as Windows.

Now that Delphi 7 has been released, Borland has *reorganized and redesignated the meanings of existing acronyms*. This can be a big source of confusion, so please pay close attention. Starting with Delphi 7 the new designations are as follows:

- CLX - CLX refers to the entire component framework included in Delphi, C++ Builder, and Kylix.
- VCL - VCL refers to the visual components that link directly to the Win32 API.
- Visual CLX - Visual CLX refers to the cross platform visual components that are based on QT are available in Delphi and Kylix.
- VCL for .net - VCL for .net refers to a new VCL that runs under .net and provides a compatibility layer for older applications, as well as additional functionality.

If you look at the new designations, I am not so sure they are consistent. I think in the future they will lead to further confusion. I think NLX (Nelix?) or NCL (Nickel?), or really anything more consistent may have been a better choice for VCL for .Net. As it stands, Visual CLX is a subset of CLX, but VCL for .net is a sibling of VCL, as is Visual CLX.

A secret decoder ring might look something like this:

- VCL --> CLX

- CLX --> Visual CLX
- Visual Parts of VCL --> VCL

Ok, so if this is all as clear as mud... On to the next topic.

20.1.3.5 VCL for .Net

VCL for .net refers to a new VCL that runs under .net and provides a compatibility layer for older applications, as well as additional functionality.

VCL for .net allows existing applications to be ported quickly as it is very similar to the Win32 VCL and CLX. This also allows for continued development of cross platform applications. This is an important feature that allows continued support of Windows installations without the .net framework, and Linux as well.

20.1.3.6 WinForms or VCL for .Net?

This is definitely an area of confusion for users - Should I use WinForms or VCL for .net to develop the GUI?

The following comparison can help you choose. A hard fast rule for all projects should not be established, but each project should be evaluated separately.

VCL for .Net	WinForms
Larger deployment size because of additional assemblies.	Smaller deployment because all assemblies are i the .net framework.
For Win32 platforms only.	Subsets available in the compact .net framework PC's. May be ported to other .net implementations
High degree of compatibility with existing code.	Requires significant changes to existing code.
Cross platform with VCL for Win32, and Visual CLX for Linux (and Windows).	.Net framework only.
More efficient in some areas.	Does not implement all the optimizations that the
Additional features and classes. This includes additional visual controls, but also things such as action lists, data access, and more.	
Full source code is available.	No source code is available.

Another possibility is this - you can mix and match. VCL for .net and WinForms are not exclusive and can be used together in the same application.

20.1.4 Additions

Some of the additions to Delphi.net are quite important in porting applications, while others are not. The items not vital to porting an application will only be covered very briefly.

20.1.4.1 Mapping to CTS

For languages to be able to work with the .net classes and other languages, they must use the CTS (Common Type System). Delphi.net could have easily made them available as additions to Delphi's types. But this would have left the situation where normal Delphi code used one set of types, and interfaces to .net used another. This would have resulted in constant copying of data back and forth, which is not the idea behind .net. This situation would be similar to what existed with COM.

To avoid this problem, the native types in Delphi.net have been mapped to types in the CTS. So when an Integer is declared, it is actually a .net Integer from the CTS. This relationship is not limited to simple types, but also extends to objects as well.

Here is a list of some mappings:

Delphi.net	.Net Common Type System
String	System.String
Variant	System.ValueType
Records	System.ValueType
Exception	System.Exception
TObject	System.Object
TComponent	System.ComponentModel.Component

20.1.4.2 Namespaces

To avoid conflicts, and also as part of the CLS (Common Language Specification), Delphi now supports namespaces. Each unit now exists within a namespace.

Thus you will now see declarations that look like these:

```
uses
  Borland.Delphi.SysUtils;

var
  GStatus: System.Label;
```

This is an important item to note as VCL for .net is namespaced and will affect your uses clauses.

20.1.4.3 Nested Types

Nested types allow type definitions to exist inside of other type definitions.

20.1.4.4 Custom Attributes

.Net does not implement RTTI as Delphi does. Instead it supports something similar called reflection. Reflection performs the same role as RTTI but functions a little differently. Reflection depends on attributes to perform some of its functions. To support this Delphi.net has extensions to declare attributes.

20.1.4.5 Other

Delphi.net also supports many new smaller additions to the language such as static class data, record inheritance, class properties, and more. Most of the enhancements relate to features of classes at the language level.

While useful and required by the CLS, they are not essential in porting applications.

20.1.5 Restrictions

Developing for Delphi.net requires several restrictions to be enforced. These restrictions are so that Delphi code can conform to the restrictions and requirements of .net.

20.1.5.1 Unsafe Items

Delphi 7 has a new designation called "Unsafe". When you compile code in Delphi 7, it will warn you about unsafe items. Unsafe items are items that are cannot be used within the .net runtime.

These warning are on by default and slow down the compiler drastically. Thus if you are compiling code that is not intended for .net, you should turn off the unsafe warnings. They produce what I call the "C++ effect". That is they make the compiler slow, and produce so many warnings that the important warnings are washed out by a high "Signal to noise ratio".

Delphi 7 can be a useful tool for evaluating code that you wish to port to .net however, and DCCIL does not produce the unsafe warnings. So the first step is to compile your code in Delphi 7 and eliminate the unsafe warnings.

Delphi separates the warnings into three types: unsafe types, unsafe code, and unsafe casts.

20.1.5.1.1 Unsafe Types

Unsafe types include the following:

- Character pointers: PChar, PWideChar, and PAnsiChar
- Untyped pointers
- Untyped var and out parameters
- File of <type>
- Real48
- Variant records (not to be confused with variants)

20.1.5.1.2 Unsafe Code

Unsafe code includes the following:

- Absolute variables
- Addr(), Ptr(), Hi(), Lo(), Swap()
- BlockRead(), and BlockWrite()
- Fail()

- GetMem(), FreeMem(), ReallocMem()
- Assembly code
- @ pointer operator

20.1.5.1.3 Unsafe Casts

Unsafe casts include the following:

- Casting an instance to a class other that is not an ancestor or descendent of the instance class type.
- Any casting of a record type

20.1.5.2 Deprecated Functionality

Several items have been deprecated as they are not compatible with .net, and thus are useless. Many of these items you will recognize from the earlier sections as they are flagged as unsafe.

- Real48 floating type. Use BCD or other math functions.
- GetMem(), FreeMem(), and ReallocMem(). Use dynamic arrays, or .net management of classes.
- BlockRead(), BlockWrite(). Use .net framework classes instead.
- Absolute directive
- Addr, and @. Use classes instead of memory blocks.
- Old Pascal type object syntax using the object keyword. Use class keyword instead.
- TVarData and direct access to variant internals. Variant semantics will be supported but not by using direct access to the internals.
- File of <type> - The size of type varies from platform to platform and thus cannot be known at compile time and thus this is not supported.
- Untyped var and out parameters. Use const parameters or ancestor class types.
- PChar. In the future Delphi.net will support PChar as unmanaged code. However unmanaged code is not recommended.
- automated and dispid directives. These directives do not apply in .net.
- asm statement - Assembly is not supported because in the .net system, code is not compiled to machine code.
- TInterfacedObject, which includes AddRef, QueryInterface, and Release.
- Dynamic aggregation - Use implements instead. Note: Implements is not implemented in the current release of DCCIL.
- ExitProc

20.1.6 Changes

Borland has invested a lot of resources in preserving compatibility as much as possible. And Delphi.net is still Delphi, but some things could not be preserved in a backwards compatible manner.

20.1.6.1 Destruction

Destruction is quite different under Delphi.net. Most code will not need to be adjusted, but it is still very important to understand how the process is different.

20.1.6.1.1 Deterministic Destruction

In a normal Delphi application destruction of objects occurs explicitly. That is, an object is destroyed when code explicitly requests its destruction using free or destroy. Destruction can also occur as part of an ownership relationship, but in the end this is a result of an explicit call to free or destroy as well on behalf of the owner. This behavior is known as deterministic destruction.

Deterministic destruction allows for more control, but is also very prone to memory leaks. It also allows for errors when a freed object is later referenced by another reference which was is not aware that the object has been destroyed.

Destruction takes care of both clean up of the object (finalization) with explicit code in the destructor, and freeing of the memory used by the object. Because the destructor takes care of both functions, Delphi programmers have come to think and treat the roles as a single one.

20.1.6.1.2 Non-Deterministic Destruction

.Net separates these the functions of finalization and memory deallocation because of how memory is managed in .net. .Net uses non-deterministic destruction to achieve this. If you have worked with interfaces, the reference counting semantics used for interfaces are similar.

Instead of explicitly telling objects when to destroy themselves, the CLR reference counts the objects. When an object is no longer used, it is marked for destruction.

20.1.6.2 Garbage Collection

.Net uses garbage collection to clean up the memory used by objects. This is the name of the process which determines which objects are no longer needed, and deallocates the memory associated with them.

The .net garbage collector is quite complex and even a basic introduction is worthy of a section, if not a complete paper of its own.

Other than deterministic versus non-deterministic destruction, garbage collection has little impact on the porting of applications. Thus for the purposes of a porting, the garbage collector shall be considered a "Magic Box" that takes care of destroying objects for you.

20.1.7 Porting Steps

Porting applications to Delphi.net for most applications will have quite a large impact and must be approached cautiously. The more object oriented the code, the easier it will be. This article cannot reduce the amount of work that will need to be done, but it will help you approach the task better prepared and informed. This will reduce the mistakes that are made, and provide for a faster implementation of the port.

20.1.7.1 Remove Unsafe Warnings

To remove unsafe warnings, load the target project into Delphi 7. With the unsafe warnings turned on, perform a build all. Delphi will produce a series of unsafe warnings. Each warning needs to be eliminated. This may easily be the biggest step in porting your application.

20.1.7.2 Unit Namespaces

Uses clauses need to be converted to use namespaces.

If the code needs to continue to run on Windows and the .net framework the units can be IFDEFed as follows. CLR is defined in Delphi.net.

```
uses
  {$IFDEF CLR}Borland.Win32.Windows{$ELSE}Windows{$ENDIF},
  {$IFDEF CLR}Borland.Delphi.SysUtils{$ELSE}SysUtils{$ENDIF},
  {$IFDEF CLR}Borland.Vcl.Forms{$ELSE}Forms{$ENDIF};
```

The namespaces can be seen. The three major ones are Borland.Win32 (Windows), Borland.Delphi (RTL) and Borland.VCL (VCL for .net).

20.1.7.3 Convert DFM's

Delphi for .net does not yet support DFM's. It will in the future, so this step is only needed if you wish to use the beta version of DCCIL.

Because DFMs's are not supported, all forms must be constructed using code. There are tools designed to perform this task that have been used with normal Delphi applications and are useful for this task.

20.1.7.4 Convert Project File

Application.CreateForm is not supported, so forms must be manually created as normal components. To set the main form of an application, set Application.MainForm before calling Application.Run.

20.1.7.5 Resolve Class Differences

During compilation there will be slight differences between VCL for .net and VCL, just as there are minor differences between VCL and Visual CLX. Each of these will need to be resolved manually or IFDEFed.

20.1.7.6 Add Luck

With luck, your project will be converted and function properly.

Porting is not a simple recompile and should be allocated proper time. Porting to .net takes a similar effort that would be needed for porting a VCL application to Visual CLX. Although the time and effort required is significant, it is a lot less than a complete rewrite, and it allows for cross platform deployment as well as code sharing if proper planning is performed before hand.

20.1.8 Credits

I have used many sources. I am sorry if I have forgotten to credit anyone. I wish to thank the following individuals: John Kaster, Brian Long, Bob Swart, Lino Tadros, Danny Thorpe, Eddie Churchill, Doychin Bondzhev, and last but not least, experimentation.

Section

About the Authors

21 About the Authors

21.1 Chad Z. Hower a.k.a Kudzu

Chad Z. Hower, a.k.a. "Kudzu" is the original author and project coordinator for Internet Direct (Indy). Indy consists of over 110 components and is included as a part of Delphi, Kylix and C++ Builder. Chad's background includes work in the employment, security, chemical, energy, trading, telecommunications, wireless, and insurance industries. Chad's area of specialty is TCP/IP networking and programming, inter-process communication, distributed computing, Internet protocols, and object-oriented programming. When not programming, he likes to cycle, kayak, hike, downhill ski, drive, and do just about anything outdoors. Chad, whose motto is *"Programming is an art form that fights back"*, also posts free articles, programs, utilities and other oddities at Kudzu World at http://www.Hower.org/Kudzu/. Chad is an American ex-patriate who currently spends his summers in St. Petersburg, Russia and his winters in Limassol, Cyprus. Chad can be reached at cpub@Hower.org.

Chad works as a Senior Developer for Atozed Computer Software Ltd. (http://www.atozedsoftware.com/)

21.2 Hadi Hariri

Hadi Hariri is a Senior Developer and Project Manager at Atozed Computer Software Ltd. (http://www.atozedsoftware.com/) and 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 and has spoken at Borland Conferences and user groups.

Index

- T -