Title: The synapse Cookbook

Abstract

Various articles about programming using synapse tcp/ip library for object pascal
Originated from the synapse cookbook wiki which is no longer online. Still partly incomplete.

Document version: 0.1
Author: R.M. Tegel, various authors
Composer: R.M. Tegel

Introduction

Most of this document is extracted from the wayback-machine due to an unfortunate miscommunication between me and my hosting. I take all the blame though.
Anyhow, since the wiki was not working as expected i learned a lot from it. Like: better not to run a wiki or make sure you have a wiki system that sends email alerts on any change made or whatever.
Also, i think a wiki is somewhat hard to organize, any index will remain another wiki page.
At least this document has a tail and an end, or maybe better: an open end.
This first version is a very quick release just to make sure there is a document. Not alll content is (re)edited or valued for sanity;P

This document is at the moment not-as-strutured as i'd like to be. Beginners may want to look at Appendix F – The tutorials. The part “Synapse in two hours” shows some elementary code, as well as gives you a kickstart in writing multi-threaded applications.

Table of contents

Table of Contents

The Socket classes

TBlockSocket

Abstract

Synapse base socket class. From this class (indirectly via TSoc ksBlockSocket?) the TTCPBlockSocket, TUDPBlockSocket? and TRawBlockSocket?? are derived.

Description

The TBlockSocket class contains all methods to read and write to network sockets. It applies to any kind of socket, like TCP and UDP.

Opening and connecting

You open a socket using the Connect or the Bind method, bind optionally followed by a Listen for listening on TCP sockets.
UDP sockets will automatically bind if Connect is called.
For outgoing TCP connections you just have to use the Connect (IP, Port) method.

Sending and receiving data

You can send and receive packets, blocks (buffers or strings) of data, lines, bytes or streams using the appropiate Send<method> or Recv<method>

procedure SendString? (Data:String) sends a string of data
fucntion RecvPacket? (TimeOut?:Integer):String receives a packet of data.

function RecvString?(Timeout:Integer):String is special.
It calls RecvTerminated? (CRLF), which in turn receives a line of data, terminated with a carriage return and a linefeed. This is very practical in commonly used line-based protocols.
For higher compatability, you can set the ConvertLineEnd? property to true:

example

```
mysock := TTCPBlockSocket.Create;
mysock.ConvertLineEnd := True;
mysock.Connect ('somehost', '110');
//This will read a line terminated by #A0 (LF) or #D0A0 (CRLF)
m := mysock.RecvString (30000); //read welcome message
m.SendString ('USER me'+CRLF); //you must send CRLF yourself
s:=m.RecvPacket (20000);
s:='PASS dontknow'+CRLF;
m.SendBuffer (pointer(s), length(s));
setlength (s, m.WaitingData);
if s<>'' then
m.RecvBuffer (@s[1], length(s));
m.CloseSocket;
m.Free;
```

WaitingData? and CanRead? are other commonly used functions.

public and published properties of TBlockSocket (synapse version 32)

```
 public
 constructor Create;
 constructor CreateAlternate(Stub: string);
 destructor Destroy; override;
 procedure CreateSocket;
 procedure CreateSocketByName(const Value: String);
 procedure CloseSocket; virtual;
 procedure AbortSocket;
 procedure Bind(IP, Port: string);
 procedure Connect(IP, Port: string); virtual;
 function SendBuffer(Buffer: Pointer; Length: Integer): Integer; virtual;
 procedure SendByte(Data: Byte); virtual;
 procedure SendString(const Data: string); virtual;
 procedure SendBlock(const Data: string); virtual;
 procedure SendStream(const Stream: TStream); virtual;
 function RecvBuffer(Buffer: Pointer; Length: Integer): Integer; virtual;
 function RecvBufferEx(Buffer: Pointer; Length: Integer;
 Timeout: Integer): Integer; virtual;
 function RecvBufferStr(Length: Integer; Timeout: Integer): String; virtual;
 function RecvByte(Timeout: Integer): Byte; virtual;
 function RecvString(Timeout: Integer): string; virtual;
 function RecvTerminated(Timeout: Integer; const Terminator: string): string;
virtual;
 function RecvPacket(Timeout: Integer): string; virtual;
 function RecvBlock(Timeout: Integer): string; virtual;
 procedure RecvStream(const Stream: TStream; Timeout: Integer); virtual;
 function PeekBuffer(Buffer: Pointer; Length: Integer): Integer; virtual;
 function PeekByte(Timeout: Integer): Byte; virtual;
 function WaitingData: Integer; virtual;
 function WaitingDataEx: Integer;
 procedure Purge;
 procedure SetLinger(Enable: Boolean; Linger: Integer);
 procedure GetSinLocal;
 procedure GetSinRemote;
 procedure GetSins;
 function SockCheck(SockResult: Integer): Integer;
 procedure ExceptCheck;
 function LocalName: string;
 procedure ResolveNameToIP(Name: string; IPList: TStrings);
 function ResolveName(Name: string): string;
 function ResolveIPToName(IP: string): string;
 function ResolvePort(Port: string): Word;
 procedure SetRemoteSin(IP, Port: string);
 function GetLocalSinIP: string; virtual;
 function GetRemoteSinIP: string; virtual;
 function GetLocalSinPort: Integer; virtual;
 function GetRemoteSinPort: Integer; virtual;
 function CanRead(Timeout: Integer): Boolean;
 function CanReadEx(Timeout: Integer): Boolean;
 function CanWrite(Timeout: Integer): Boolean;
 function SendBufferTo(Buffer: Pointer; Length: Integer): Integer; virtual;
 function RecvBufferFrom(Buffer: Pointer; Length: Integer): Integer; virtual;
 function GroupCanRead(const SocketList: TList; Timeout: Integer;
 const CanReadList: TList): Boolean;
 procedure EnableReuse(Value: Boolean);
 procedure SetTimeout(Timeout: Integer);
 procedure SetSendTimeout(Timeout: Integer);
 procedure SetRecvTimeout(Timeout: Integer);
 function StrToIP6(const value: string): TSockAddrIn6;
 function IP6ToStr(const value: TSockAddrIn6): string;
 function GetSocketType: integer; Virtual;
 function GetSocketProtocol: integer; Virtual;
 property WSAData: TWSADATA read FWsaData;
 property LocalSin: TVarSin read FLocalSin write FLocalSin;
```

```
 property RemoteSin: TVarSin read FRemoteSin write FRemoteSin;
 published
 class function GetErrorDesc(ErrorCode: Integer): string;
 property Socket: TSocket read FSocket write SetSocket;
 property LastError: Integer read FLastError;
 property LastErrorDesc: string read FLastErrorDesc;
 property LineBuffer: string read FBuffer write FBuffer;
 property RaiseExcept: Boolean read FRaiseExcept write FRaiseExcept;
 property SizeRecvBuffer: Integer read GetSizeRecvBuffer write
SetSizeRecvBuffer;
 property SizeSendBuffer: Integer read GetSizeSendBuffer write
SetSizeSendBuffer;
 property NonBlockMode: Boolean read FNonBlockMode Write SetNonBlockMode;
 property MaxLineLength: Integer read FMaxLineLength Write FMaxLineLength;
 property MaxSendBandwidth: Integer read FMaxSendBandwidth Write
FMaxSendBandwidth;
 property MaxRecvBandwidth: Integer read FMaxRecvBandwidth Write
FMaxRecvBandwidth;
 property MaxBandwidth: Integer Write SetBandwidth;
 property ConvertLineEnd: Boolean read FConvertLineEnd Write FConvertLineEnd;
 property TTL: Integer read GetTTL Write SetTTL;
 property Family: TSocketFamily read FFamily Write SetFamily;
 property PreferIP4: Boolean read FPreferIP4 Write FPreferIP4;
 property IP6used: Boolean read FIP6used;
 property InterPacketTimeout: Boolean read FInterPacketTimeout Write
FInterPacketTimeout;
 property RecvCounter: Integer read FRecvCounter;
 property SendCounter: Integer read FSendCounter;
 property OnStatus: THookSocketStatus read FOnStatus write FOnStatus;
 property OnReadFilter: THookDataFilter read FOnReadFilter write FOnReadFilter;
 property OnWriteFilter: THookDataFilter read FOnWriteFilter write
FOnWriteFilter;
 property OnCreateSocket: THookCreateSocket read FOnCreateSocket write
FOnCreateSocket;
 end;
```

TTCPBlockSocket

TTCPBlockSocket provides access to the (TCP protocol)?. *See also: TBlockSocket, SetLinger?*

Unit

(blcksock)?

Hierarchy

TObject - TBlockSocket - TSocksBlockSocket - TTCPBlockSocket

Description

The TTCPBlockSocket is used to read and write information over a synchronous TCP connection. Although TTCPBlockSocket has limited support for asynchronous connections, Synapse is optimized for synchronous operation.

A few notable features are:

- IPv4 and IPv6 support
- Secure SSL/TLS (SSL2, SSL3 and TLS) support
- SOCKS5, SOCKS4/4a proxy support
- TCP through HTTP proxy tunnel

Example

```
D := TTCPBlockSocket.Create;
D.Connect ('localhost', '80');
if D.LastError = 0 then
begin
D.SendString ('HEAD / HTTP/1.0'#13#10#13#10);
s := D.RecvString (20000);
ShowMessage (s);
end;
D.Free;
```

Related

TBlockSocket, TSocksBlockSocket, (TCP)?, SetLinger?.

TUDPBlockSocket

Abstract
TUDPBlockSocket is the Synapse class for sending and receiving UDP datagrams.
http://www.ararat.cz/synapse/docs/help/TUDPBlockSocket.html

Q. Is it necessary to create a thread when using udp sockets?
A. No, in contrary to tcp/ip, you can use blocking udp in you main thread. Not highly recommended, but very well possible. Sending data costs as much time as getting the packet out of the network adapter, for reading a datagram, you can just specify a timeout of 0 ms, and check wether or not data was available.

Example

```
procedure TForm1.Button2Click(Sender: TObject);
var U:TUDPBlockSocket;
begin
U:=TRawUDPBlockSocket?.Create;
U.Connect ('192.168.0.20', '7');
ShowMessage? (inttostr(U.SendBuffer? (PChar('hello'),5))+' bytes send');
U.Free;
end;
```

Another example: udp server

TSocksBlockSocket

Abstract
Synapse socket class with socks support. Derived from TBlockSocket. From this class the TTCPBlockSocket, TUDPBlockSocket? are derived.

Fill in the socks proxy info (host, port, type (socks4 or socks5)) to add socks server proxy support for TCP and UDP protocols.

Synapse Protocol Classes

TPOP3Send

Abstract

TPOP3Send is a class to fetch email messages from a POP3 server.
Unit: pop3send.pas

Example

```
uses pop3send, mimemess;
var mailid, i: Integer;
 uid: String;
 FUIDL: TStrings;
 FPOPBox: TPOP3Send;
 Mime: TMimeMess;
//..//
 FUIDL := TStringList.Create;
 FPOPBox := TPOP3Send.Create;
 FPOPBox.TargetHost := FHostName;
 FPOPBox.Username := FUser;
 FPOPBox.Password := FPass;
 if FPOPBox.Login then
 begin
 FPOPBox.Stat;
 if FPOPBox.Uidl (0) then
 UIDL.Assign (FPOPBox.FullResult)
 else
 UIDL.Clear;
 for i:=0 to UIDL.Count -1 do
 begin
 //mailid should be (i + 1)
 mailid := StrToIntDef(Copy (S,1,pos(' ',S)-1), -1);
 //you can use UID to identify a message across sessions
 UID:=Copy (S, pos(' ',s)+1, maxint);
 FPopBox.Retr(mailid);
 //if you only need message headers use this:
 // FPOPBox.Top (mailid,0);
 Mime.Clear;
 Mime.Lines.Assign (FPOPBox.FullResult);
 Mime.DecodeMessage;
 //do with mime whatever you like
 //like presenting it to the user
 end;
 FPPOPBox.LogOut;
 end
 else
 begin
 //failed to logon
 end;
 FPopBox.Free;
 FPOP3.Free;
```

LDAP Example

```
 LLdap:= TLDAPSend.Create;
 LList:= TStringList.Create;
 try
 LLdap.TargetHost:= cmbHost.Text;
 LLdap.TargetPort:= cmbPort.Text;
 LLdap.Username:= cmbUserDN.Text;
 LLdap.Password:= cmbPassword.Text;
 LLdap.Version:= cmbVersion.ItemIndex+2;
 try
 if ( LLdap.Login ) then begin
 if ( LLdap.Bind ) then begin
 if ( LLdap.Search(cmbBaseDN.Text, False, fldQuery.Text,
LList) ) then begin
 for j:= 0 to LLdap.SearchResult.Count-1 do begin
 for k:= 0 to LLdap.SearchResult[j].Attributes.Count-1 do begin
 //get the value of MYATTRIBUTE
 if LLDAP.SearchResult.Items[j].Attributes.Items[k].AttributeName =
'MYATTRIBUTE' then begin
 Result := Trim(LLDAP.SearchResult.Items[j].Attributes.Items[k].GetText);
 end;
 end;
 end;
 end
 else begin
 end;
 end
 else begin
 end;
 end;
 except
 end;
 finally
 FreeAndNil(LList);
 FreeAndNil(LLdap);
 end;
```

by Lubos.

Visual Synapse

Visual Synapse

Description

Visual synapse is a wrapper library around the synapse libraries. You can install it as a set of components.

Visual synapse is *not* a replacement for synapse libraries. It would take a tremendous amount of work to put synapse functionality in a set of non-visual components. Also, it would not be practical in most circumstances. So, only protocols that let them pour quite well in components are encapsulated by the visual synapse library.

current state

- Visual synapse is now in development state by Rene. Beta 0.2 is the first official beta release. 0.21 is the current release.

- Visual Synapse has now got it's place at sourceforge:

http://www.sourceforge.net/projects/visualsynapse

- I'm busy with the Visual Server components. The HTTP server component is almost ready for a beta release. Hang on!

License

Visual Synapse is currently released under Artistic License conditions. It may get released under Modified Artistic License or Mozilla Public License in the future.

Downloads

beta 0.2 can be downloaded from here:
http://synapse.wiki.dubaron.com/files/visualsynapse.pas
dcr file with some images (put it in the same directory):
http://synapse.wiki.dubaron.com/files/VisualSynapse.dcr

Help needed

If somebody finds the time, please help to

- Find bugs
- Implement the FTP, TFTP, POP3 or IMAP protocol
- Design graphics for the component palette
- Write or update the documentation
- Suggestions for modification/improvement welcome

Classes provided by VisualSynapse

- TVisualSynapse - abstract base class for the components
    - TVisualHTTP
    - TVisualDNS
    - TSendMail
    - TVisualUDP
    - TVisualTCP
    - TVisualICMP
    - TIPHelper
- TVisualThread? - abstract base class for the threads
- TJob - base class for job enqueueing between component and threads

Features

- Complete multi-threading implementation in a single component. For example, one single TVisualHTTP component can retrieve unlimited number of http documents at the same time.
- safe multithreading.
- no exceptions, but a error callback
- onprogress callbacks helps you track the amount of data
- bandwith throttle. synapse supports bandwith throttle. this is partly implemented in beta 0.2 release of visual synapse.

Supported protocols

- HTTP
- PING
- TRACEROUTE
- TCP
- UDP
- DNS
- SMTP

New in release 0.2

New is the TSendMail component.

Working with socks

Put a TSocksProxyInfo? component on the form and fill in the parameters.
With one of the protocol implementations (the other components) click the socksproxyinfo property and select the cooresponding socksinfo component. You can share the socksproxyinfo component among the other components.

Multi threading

In most cases, it is not needed to add a new component for each connection you want to make. The components are able to run several threads as one. You can limit the number of threads by setting

the MaxThreads property. If more jobs are reuqested than threads are available, the jobs will stay queued until some thread has finished it's previous job.

What protocols can be encapsulated by visual synapse in the future

- FTP
- TCP server

Which protocols are impractical to implement

- POP3
- IMAP
- MIME (included in SMTP)

Visual synapse explitely is not a full implementation/wrapper of synapse libraries. Also, you are strongly advised to use synapse libraries themselves.
The components are only here for convenienve in light-weight network applications.

Added functionality is a encapsulation of the IP helper API.

About the VisualSynapse component structure:

- all classes inherit from the base class TVisualSynapse
- jobs are enqueued by the main thread and executed by the client thread

RawIP

RawIP is a pascal unit that creates some TBlockSocket descendants that allow to sniff a network or send UDP packets with spoofed addresses.

This unit demonstrates how to set up winsock 2 to allow raw packet sending and receiving.

you can download the unit from http://synapse.wiki.dubaron.com/files/rawIP.pas

An example application using rawip.pas is Easy Sniffer

In this unit you can find these classes:
* TSniffingSocket for receiving all network packets
* TRawUDPSocket example of sending raw IP headers, this implementation can spoof the sender.
* TSniffer object, that encapsulates a sniffer thread.
* TIPHelper implementation of the IP helper api iphlpapi.dll

To use with synapse, you must modify blcksock.pas a little:

> FRemoteSin? as declared in blcksock.pas must be moved from private to protected in order for TRawUDPSocket unit to use it.

Usefull links:

Sending raw sockets:
http://msdn.microsoft.com/library/en-us/winsock/winsock/tcp_ip_raw_sockets_2.asp
Socket options:
http://msdn.microsoft.com/library/en-us/winsock/winsock/tcp_ip_socket_options_2.asp
changes between wsock32.dll and ws2_32.dll:
http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/Q257/4/60.ASP&NoWebContent=1

sniffing:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/wsaioctl_2.asp
http://www.securityfriday.com/Topics/caputure_packet.html

Raw code version 0.1 - this code may be subject to changes:

```
unit rawIP;
{ Version 0.1
 This unit is explitely designed for windows. Sending raw sockets may succeed on
linux.
 The sniffing socket is solely for windows 2000/XP and needs administrator
rights to run.
 Based on synapse blocking sockets are two decsendants:
 *TSniffingSocket for receiving all network packets
 *TRawUDPSocket example of sending raw IP headers, this implementation can spoof
the sender.
 //FRemoteSin as declared in blcksock.pas must be moved from private to
 //protected in order for TRawUDPSocket unit to use it.
 Also, there is a
 *TSniffer object, that encapsulates a sniffer thread.
 for system information from ip helper api, there is:
 *TIPHelper
 see docs on synapse.wiki.dubaron.com for how to use.
```

```
 licensed under synapse compatible BSD license
 Author:
 Rene Tegel rene@dubaron.com
 release 0.1 augustus 17-19 2003.
}
interface
uses Windows, classes, Sysutils, winsock,
 blcksock, synsock, dialogs;
{relevant RFC's:
 IP: RFC 791 IP
 950 subnetting
 919 broadcasting
 922 broadcastinf and subnets
 792 ICMP
 1112 Multicasting
 UDP: RFC 768
 TCP: RFC 793
 IP, ARP and FDDI: RFC 1390
 ARP and Ethernet: RFC 826
 IP and Ethernet: RFC 894
 IP and IEEE 802: RFC 1042
 PPP: RFC 1661 Point-to-Point protocol
 RFC 1662 PPP and HDLC framing
}
//winsock 2 declarations
const SIO_RCVALL = $98000001;
//raw IP packet related constants:
const //Ethernet header flags:
 EthernetType2 = $0800;
 EthernetTypeARP = $0806;
 //IP header flages
 FlagUnused = 1 shl 15;
 FlagDontFragment = 1 shl 14;
 FlagMore = 1 shl 13;
 //ICMP predefined protocol constants:
 ICMP_Echo_Reply = 0;
 ICMP_Dest_Unreachable = 3;
 ICMP_Source_Quench = 4;
 ICMP_Redirect = 5;
 ICMP_Echo_Request = 8;
 ICMP_Router_Advertisement=9;
 ICMP_Router_Sollicitation = 10;
 ICMP_Time_Exceeded = 11;
 ICMP_Parameter_Problem = 12;
 ICMP_Timestamp_Request = 13;
 ICMP_Timestamp_Reply = 14;
 ICMP_Information_Request = 15;
 ICMP_Information_Reply = 16;
 ICMP_Address_Mask_Request = 17;
 ICMP_Address_Mask_Reply = 18;
 ICMP_TraceRoute = 30;
 //TCP header flags:
 TCP_FIN = 1; //Connection control
 TCP_SYN = 2; //Synchronize, connection control, syncing sequence numbers
 TCP_RST = 4; //RESET, end of connection
 TCP_PSH = 8; //PUSH
 TCP_ACK = 16; //Acknowledgement number is valid
 TCP_URG = 32; //Urgent pointer is valid
// IP / TCP / UDP / ICMP / ARP header structures:
//only IP and UDP headers are tested.
//for other structures i cannot guarantee they are correct.
type Byte6 = array[0..5] of byte;
 Byte3 = array[0..2] of byte;
 TIPHeader = packed record
```

```
 case Integer of
 0: (
 VerLen: Byte;
 TOS: Byte;
 TotalLen: Word;
 Identifer: Word;
 FragOffsets: Word;
 TTL: Byte;
 Protocol: Byte;
 CheckSum: Word;
 SourceIp: DWORD;
 DestIp: DWORD;
 // Options: DWORD; //no options by default, header size 5 DWords
 );
 1: (Raw: Array[0..9] of Word);
 end;
 PIPHeader = ^TIPHeader;
 TArpHeader = packed record
 //Ethernet type typically $0806
 Hardware: Word;
 Protocol: Word;
 HardwareAddressLength:byte;
 ProtocolLength: byte;
 Operation:Word;
 SenderHWAddress:Byte6;
 SenderIPAddress:DWord;
 TargetHWAddress:Byte6;
 TargetIPAddress:DWord;
 end;
 TEthernetHeader = packed record
 DestMacAddress:Byte6; //leave $FF FFFF for broadcasts
 SourceMacAddress:Byte6;
 ProtocolType:Word; //mostly is EthernetType2
 end;
{ An EthernetPacket typically looks as follows:
 * TEthernetHeader
 * TIPHeader
 * TCP or UDP header
 * User data (TCP/UDP)
 * CRC checksum, DWord. wrapped by the ethernet protocol.
 so there is no real use for checksumming inside the user data.
}
 TICMPHeader = packed record
 ICMPtype:byte;
 code:byte;
 Checksum:Word;
 end;
 TICMPPacket = packed record //??
 IPHeader:TIPHeader;
 ICMPHeader:TICMPHeader;
 Data:Array[0..1499] of byte;
 end;
 TTCPHeader = packed record
 SourcePort:Word;
 DestPort:Word;
 SequenceNumber:DWord;
 AcknowledgementNumber:DWord;
 Offset:Byte; //only left 4 bits. Header length in 32-bit segments
 Flags:Byte;
 Window:Word;
 Checksum:Word; //includes speudo header instead of TCP header.
 UrgentPointer:Word;
 {Optionally:
 Options:byte3; //MSS (Maximum Segment Size) at connection startup
```

```
 Padding:byte;
 }
 end;
 TTCPPacket = packed record
 IPHeader:TIPHeader;
 TCPHeader:TTCPHeader;
 Data:Array[0..32767] of byte;
 end;
 TUDPHeader = packed record
 case Integer of
 0: (
 SourcePort,
 DestPort : Word; //why why why a Dword ???
 Length,
 Checksum:word;
 );
 1: (
 Raw:Array[0..3] of Word;
 );
 end;
 PUDPHeader = ^TUDPHeader;
// Helper functions from the ip helper api (iphlpapi.dll)
// Next types extracted from whirwater:
//
http://www.whirlwater.com/frames.php?http://www.whirlwater.com/information/2001/
windows2000/usingtheiphelperapi.html
// thanx for coverting type definitions fellows
const
 MAX_HOSTNAME_LEN = 128; { from IPTYPES.H }
 MAX_DOMAIN_NAME_LEN = 128;
 MAX_SCOPE_ID_LEN = 256;
 MAX_ADAPTER_NAME_LENGTH = 256;
 MAX_ADAPTER_DESCRIPTION_LENGTH = 128;
 MAX_ADAPTER_ADDRESS_LENGTH = 8;
type
 TIPAddressString = Array[0..4*4-1] of Char;
 PIPAddrString = ^TIPAddrString;
 TIPAddrString = Record
 Next : PIPAddrString;
 IPAddress : TIPAddressString;
 IPMask : TIPAddressString;
 Context : Integer;
 End;
 PFixedInfo = ^TFixedInfo;
 TFixedInfo = Record { FIXED_INFO }
 case integer of
 0: (
 HostName : Array[0..MAX_HOSTNAME_LEN+3] of Char;
 DomainName : Array[0..MAX_DOMAIN_NAME_LEN+3] of Char;
 CurrentDNSServer : PIPAddrString;
 DNSServerList : TIPAddrString;
 NodeType : Integer;
 ScopeId : Array[0..MAX_SCOPE_ID_LEN+3] of Char;
 EnableRouting : Integer;
 EnableProxy : Integer;
 EnableDNS : Integer;
 );
 1: (A:Array[0..2047] of byte);
 End;
 PIPAdapterInfo = ^TIPAdapterInfo;
 TIPAdapterInfo = Record { IP_ADAPTER_INFO }
 Next : PIPAdapterInfo;
 ComboIndex : Integer;
 AdapterName : Array[0..MAX_ADAPTER_NAME_LENGTH+3] of Char;
```

```
 Description : Array[0..MAX_ADAPTER_DESCRIPTION_LENGTH+3] of Char;
 AddressLength : Integer;
 Address : Array[1..MAX_ADAPTER_ADDRESS_LENGTH] of Byte;
 Index : Integer;
 _Type : Integer;
 DHCPEnabled : Integer;
 CurrentIPAddress : PIPAddrString;
 IPAddressList : TIPAddrString;
 GatewayList : TIPAddrString;
 DHCPServer : TIPAddrString;
 HaveWINS : Bool;
 PrimaryWINSServer : TIPAddrString;
 SecondaryWINSServer : TIPAddrString;
 LeaseObtained : Integer;
 LeaseExpires : Integer;
 End;
//now some examples of support functions.
//maybo todo: create IPHelper object from this.
//get primary IP of each adapter on the system:
function GetAdapters(Strings:TStrings):Boolean;
//list all available IP's of this system:
//function getLocalIPList(Strings:TStrings):Boolean;
//convert local adapter IP address to mac address:
//function GetMacAddress (Adapter:String):String;
//lists actual systems dns servers:
//function ListDNSservers(Strings:TStrings):Boolean;
//externals:
function GetNetworkParams(FI : PFixedInfo; Var BufLen : Integer) : Integer;
 StdCall; External 'iphlpapi.dll' Name 'GetNetworkParams';
function GetAdaptersInfo(AI : PIPAdapterInfo; Var BufLen : Integer) : Integer;
 StdCall; External 'iphlpapi.dll' Name 'GetAdaptersInfo'
////////////////////////////////////////////////////////////////
 //Classes that implement protocols on top of raw IP:
type
 TRawUDPBlockSocket = class (TBlockSocket)
 public
 IPHeader:TIPHeader;
 UDPHeader:TUDPHeader;
 Data:Array[0..2047] of byte;
 procedure CreateSocket;// override;
 procedure CalcUDPChecksum;
 procedure Connect(IP, Port: string); override;
 procedure SetFrom(IP, Port: string);
 function SendBuffer(Buffer: Pointer; Length: Integer): Integer; override;
 end;
 TSniffingSocket = class (TBlockSocket)
 FAdapterIP:String;
 procedure CreateSocket;
 end;
 TOnPacketSniffed = procedure (Sender:TObject; Data:String) of Object;
 TSnifferThread = class (TThread)
 FData:String;
 FOwner:TObject;
 FSocket:TSniffingSocket;
 procedure SyncPacket;
 procedure Execute; override;
 end;
 TSniffer = class (TObject) //make it component if you like
 private
 FOnPacket:TOnPacketSniffed;
 protected
 FActive:Boolean;
 public
 FSniffer:TSnifferThread;
```

```
 FAdapter:String;
 procedure SetActive(Value:Boolean);
 constructor Create;
 destructor Destroy;
 published
 property Adapter:String read FAdapter write FAdapter;
 property Active:Boolean read FActive write SetActive;
 property OnPacketSniffed:TOnPacketSniffed read FOnPacket write FOnPacket;
 end;
 //ip helper interface
 TIPHelper = class (TObject) //make that component if you like
 //After construction, these strings will be created and filled
 //system wide settings:
 HostName : String;
 DomainName : String;
 CurrentDNSServer : String;
 DNSServerList : TStrings;
 NodeType : Integer;
 ScopeId : String;
 EnableRouting : Boolean;
 EnableProxy : Boolean;
 EnableDNS : Boolean;
 //Filled per adapter:
 DNSServers:TStrings;
 AdapterIPs:TStrings;
 AdapterNames:TStrings;
 AdapterDescriptions:TStrings;
 AdapterMACs:TStrings;
 DHCPServers:TStrings;
 GateWays:TStrings;
 CurrentIPs:TStrings;
 CurrentMasks:TStrings;
// PrimaryIPs:TStrings;
// PrimaryMasks:TStrings;
 //LeaseObtained:TList
 //LeaseExpired:TList
 //multiples filled per adapter
 AllIPS:TStrings;
 AllMasks:TStrings;
 constructor Create;
 destructor Destroy;
 end;
implementation
//support functions:
constructor TIPHelper.Create;
var Data:String;
 l:Integer;
 PInfo:PIPAdapterInfo;
 PIP : PIPAddrString;
 NWInfo:PFixedInfo;
 M:String;
 i:Integer;
 procedure AddrToStrings (P:PIPAddrString; IP:TStrings; Mask:TStrings);
 begin
 while P<>nil do
 begin
 if Assigned (IP) then IP.Add(P^.IPAddress);
 if Assigned (Mask) then Mask.Add(P^.IPMask);
 P := P^.next;
 end;
 end;
begin
 inherited;
 DNSServerList:=TStringList.Create;
```

```
 DNSServers:=TStringList.Create;
 AdapterIPs:=TStringList.Create;
 AdapterNames:=TStringList.Create;
 AdapterDescriptions:=TStringList.Create;
 AdapterMACs:=TStringList.Create;
 DHCPServers:=TStringList.Create;
 GateWays:=TStringList.Create;
 CurrentIPs:=TStringList.Create;
 CurrentMasks:=TStringList.Create;
// PrimaryIPs:=TStringList.Create;
// PrimaryMasks:=TStringList.Create;
 //LeaseObtained:TList
 //LeaseExpired:TList
 //multiples filled per adapter
 AllIPS:=TStringList.Create;
 AllMasks:=TStringList.Create;
 //Now fill structures
 //Fill Strings with an array of adapters
 SetLength (Data, 8192); //arbritrary, increase if you expect loads of adapters.
 PInfo := @Data[1];
 l:=length(Data);
 if 0 = GetAdaptersInfo (PInfo, l) then
 //now PInfo contains list of adapters:
 while (PInfo<>nil) and
 (Integer(PInfo)<=Integer(@Data[Length(Data)])-SizeOf(TIPAdapterInfo)) do
 begin
 AdapterNames.Add (PInfo^.AdapterName);
 AdapterDescriptions.Add (PInfo^.Description);
 M:='';
 for i:= 1 to PInfo^.AddressLength do
 M:=M+IntToHex (byte(PInfo^.Address[i]), 2);
 AdapterMacs.Add (M);
 if Assigned (PInfo^.CurrentIPAddress) then
 begin
 CurrentIPs.Add(String(PInfo^.CurrentIPAddress^.IPAddress));
 CurrentMasks.Add(PInfo^.CurrentIPAddress^.IPMask);
 end;
 AddrToStrings (@PInfo^.GatewayList, GateWays, nil);
 AddrToStrings (@PInfo^.DHCPServer, DHCPServers, nil);
 AddrToStrings (@PInfo^.IPAddressList, AllIPs, AllMasks);
 PInfo := PInfo^.Next;
 end;
 //Now fill system-wide settigs:
 NWInfo := @Data[1];
 if 0=GetNetworkParams(NWInfo, l) then
 begin
 Hostname := NWInfo^.HostName;
 DomainName := NWInfo^.DomainName;
 if Assigned (NWInfo^.CurrentDNSServer) then
 CurrentDNSServer := NWInfo^.CurrentDNSServer^.IPAddress;
 AddrToStrings (@NWINfo^.DNSServerList, DNSServers, nil);
 EnableRouting := boolean (NWInfo^.EnableRouting);
 EnableProxy := boolean (NWInfo^.EnableProxy);
 EnableDNS := boolean(NWInfo^.EnableDNS);
 ScopeID := NWInfo^.ScopeId;
 NodeType := NWInfo^.NodeType;
 end;
end;
destructor TIPHelper.Destroy;
begin
 DNSServerList.Free;
 DNSServers.Free;
 AdapterIPs.Free;
 AdapterNames.Free;
```

```
 AdapterDescriptions.Free;
 AdapterMACs.Free;
 DHCPServers.Free;
 GateWays.Free;
 CurrentIPs.Free;
 CurrentMasks.Free;
// PrimaryIPs.Free;
// PrimaryMasks.Free;
 //LeaseObtained.Free
 //LeaseExpired.Free
 AllIPS.Free;
 AllMasks.Free;
 inherited;
end;
function getAdapters;
var Data:String;
 l:Integer;
 PInfo:PIPAdapterInfo;
 PIP : PIPAddrString;
begin
 //Fill Strings with an array of adapters
 Result := False;
 if (Strings=nil) or not (Strings is TStrings) then
 exit;
 Strings.Clear;
 SetLength (Data, 8192); //arbritrary, increase if you expect loads of adapters.
 PInfo := @Data[1];
 l:=length(Data);
 if 0 <> GetAdaptersInfo (PInfo, l) then
 exit;
 //now PInfo contains list of adapters:
 while (PInfo<>nil) and
 (Integer(PInfo)<=Integer(@Data[Length(Data)])-SizeOf(TIPAdapterInfo)) do
 begin
 PIP := @PInfo^.IPAddressList;
 while PIP<>nil do
 begin
 Strings.Add (PIP^.IPAddress);
 PIP := PIP^.Next;
 Result := True;
 end;
 PInfo := PInfo^.Next;
 end;
end;
procedure TRawUDPBlockSocket.CreateSocket;
var c:Integer;
 Sin: TVarSin;
 IP:String;
 Port:String;
begin
 FSocket := synsock.Socket(AF_INET, SOCK_RAW, IPPROTO_IP);
 inherited CreateSocket;
 c:=1;
 setsockopt(FSocket, 0{SOL_SOCKET}, IP_HDRINCL, @c, sizeof(c));
 //fill header info
 with IPHeader do
 begin
 Protocol := 17; //UDP
 TTL := 128;
 VerLen := (4 shl 4) or 5;
 end;
end;
procedure TRawUDPBlockSocket.CalcUDPChecksum;
//calc UDP checksum
```

```
var checksum:Integer;
 i,l,m:Integer;
begin
 // see rfc 768;
 // http://www.faqs.org/rfcs/rfc768.html
 checksum := 0;
 l := ntohs(UDPHeader.Length) - SizeOf(TUDPHeader); //data length
 m := l div 2; //see if padding is needed
 if (l mod 2)<>0 then
 begin
 Data[l] := 0; //add padding zero
 inc (m);
 end;
 //checksum the data:
 for i:=0 to m-1 do
 Checksum := Checksum - ((Data[i*2] shl 8) + Data[i*2+1])-2;
 //pseudo headers, source+dest:
 for i:= 8 to 9 do
 Checksum := Checksum - IPHeader.Raw[i] -1;
 //pseudo headers: proto + udplength
 Checksum := Checksum - IPHeader.Protocol - UDPHeader.Length -2;
 //take the one's complement:
 Checksum := - Checksum - 1;
 //now make 16 bits from 32 bits:
 while (Checksum shr 16)>0 do
 Checksum := ((Checksum shr 16) and $FFFF) or (Checksum and $FFFF);
 UDPHeader.Checksum := 0; //Checksum; it aint working yet.
end;
procedure TRawUDPBlockSocket.Connect;
type
 pu_long = ^u_long;
var HostEnt:PHostEnt;
begin
 //inherited connect is of no use here, since we deal with raw sockets.
 //fill the IP header structure
 inherited Connect (IP, Port); //fill sins
 if IP = cBroadCast then
 IPHeader.DestIP := INADDR_BROADCAST
 else
 begin
 IPHeader.DestIP := synsock.inet_addr(PChar(IP));
 if IPHeader.DestIP = INADDR_NONE then
 begin
 HostEnt := synsock.GetHostByName(PChar(IP));
 if HostEnt <> nil then
 IPHeader.DestIP := u_long(Pu_long(HostEnt^.h_addr_list^)^);
 end;
 end;
 UDPHeader.DestPort := htons (StrToIntDef (Port, 0));
end;
procedure TRawUDPBlockSocket.SetFrom;
type pu_long = ^ulong;
var HostEnt:PHostEnt;
begin
 if IP = cBroadCast then
 IPHeader.SourceIP := INADDR_BROADCAST
 else
 begin
 IPHeader.SourceIP := synsock.inet_addr(PChar(IP));
 if IPHeader.SourceIP = INADDR_NONE then
 begin
 HostEnt := synsock.GetHostByName(PChar(IP));
 if HostEnt <> nil then
 IPHeader.SourceIP := u_long(Pu_long(HostEnt^.h_addr_list^)^);
```

```
 end;
 end;
 UDPHeader.SourcePort := htons(StrToIntDef (Port, 0));
end;
function TRawUDPBlockSocket.SendBuffer;
var P:String; //the actual packet
 d:TSockAddr;
 l:Integer;
begin
 if Length>=high(Data) then
 begin
 Result := -1;
 exit;
 end;
 //set header checksum
 IPHeader.TotalLen := htons(SizeOf(TIPHeader)+SizeOf(TUDPHeader)+Length);
 IPHeader.Identifer := gettickcount mod $FFFF; //0 also allowed, then kernel
fills it.
 IPHeader.FragOffsets := $00; //htons(FlagDontFragment);
// CalcHeaderChecksum; //don't, kernel does this!
 //move data
 move (Buffer^, Data[0], Length);
 //set udp checksum
 UDPHeader.Length := htons(SizeOf(TUDPHeader)+Length);
 CalcUDPChecksum; //you can leave it zero if you like: UDPHeader.Checksum := 0;
 //move to buffer,
 //setlength of total IP packet:
 SetLength (P, SizeOf(TIPHeader)+SizeOf(TUDPHeader)+Length);
 //move IP header:
 move (IPHeader.Raw[0], Pointer(P)^, SizeOf (TIPHeader));
 //move IP data, in this case: UDP header:
 move (UDPHeader.Raw[0], P[1+SizeOf(TIPHeader)], SizeOf (TUDPHeader));
 //move UDP data:
 move (Data[0], P[1+SizeOf(TIPHeader)+SizeOf(TUDPHeader)], Length);
 //send data
 l:=system.Length(P);
// Connect (IPHeader.DestIP, IPHeader.Port);
 Result :=
 sendto (FSocket,
 P[1],
 l,
 0, //flags
 PSockAddr(@FRemoteSin), //we need this filled for the kernel to send.
 SizeOf(FRemoteSin));
end;
procedure TSniffingSocket.CreateSocket;
var c,d,l:Integer;
 F:TStrings;
 Sin: TVarSin;
begin
 FSocket := synsock.Socket(AF_INET, SOCK_RAW, IPPROTO_IP);
// no inherited CreateSocket here.
 c:=1;
 if FAdapterIP = '' then
 begin
 //fetch adapterIP
 //get default (first listed) adapter:
 F:=TStringList.Create;
 if (not GetAdapters(F)) or (F.Count=0) then
 exit; //don't try further, no use without IP.
 FAdapterIP := F[0];
 end;
 SetSin(Sin, FAdapterIP, '0');
 SockCheck(synsock.Bind(FSocket, @Sin, SizeOfVarSin(Sin)));
```

```
// setsockopt(FSocket, 0{SOL_SOCKET}, IP_HDRINCL, @c, sizeof(c)); //not
necessary
 c := 1;
 d:=0;
 FLastError := WSAIoctl (FSocket, SIO_RCVALL, @c, SizeOf(c), @d, SizeOf(d),@l ,
nil, nil);
end;
procedure TSnifferThread.SyncPacket;
begin
 try
 if Assigned (TSniffer(FOwner).FOnPacket) then
 TSniffer(FOwner).FOnPacket (FOwner, FData);
 except //build-in safety, stop sniffing if anything fails:
 FOwner := nil;
 Terminate;
 end;
end;
procedure TSnifferThread.Execute;
begin
 FSocket.CreateSocket;
 while not Terminated do
 begin
 if (FSocket.WaitingData>0) and (FSocket.WaitingData<1000000) then
 begin
 SetLength (FData, FSocket.WaitingData);
 SetLength (FData,
 FSocket.RecvBuffer (@FData[1], length(FData)) );
 if FData<>'' then
 Synchronize (SyncPacket);
 end
 else
 sleep(0);
 end;
end;
constructor TSniffer.Create;
begin
 inherited;
end;
destructor TSniffer.Destroy;
begin
 Active := False;
 inherited;
end;
procedure TSniffer.SetActive;
begin
 if Value = FActive then
 exit;
 if Value then
 begin
 FSniffer := TSnifferThread.Create (True);
 FSniffer.FSocket := TSniffingSocket.Create;
 FSniffer.FSocket.FAdapterIP := FAdapter;
 FSniffer.FOwner := Self;
 FSniffer.FreeOnTerminate := True;
 FSniffer.Resume;
 end
 else
 begin
 FSniffer.Terminate;
 FSniffer.WaitFor;
 FSniffer.Free;
 end;
 FActive := Value;
end;
```

```
end.
{
procedure TRawBlockSocket.CalcHeaderChecksum;
//funny, we don't need it. the kernel does.
var i:Integer;
 checksum:DWord;
begin
 //calc the IP header checksum:
 IPHeader.CheckSum := htons(0); //fill with network zero, since it gets included
in packet as well.
 checksum := 0;
 for i:=0 to 9 do
 checksum := checksum + IPHeader.Raw[i];
 //shift and add high bytes:
 checksum := (checksum shr 16) + (checksum and $FFFF);
 //and add optional carry bit:
 inc (checksum, checksum shr 16);
// checksum := 1+ (not checksum); //complement
 checksum:=DWord(-Integer(checksum));
 IPHeader.CheckSum := htons(Checksum and $FFFF);
end;
}
```

Appendix A. Definitions

Borland

Producer of software development tools.

homepage http://www.borland.com

creates pascal, c, c++ and c# compilers for windows and linux (x86)

free versions:

- Delphi 6 Personal edition
- Delphi 7 Personal edition
- Kylix 1,2,3 Personal edition
- C 5.0 compiler

Freepascal

Freepascal is a cross-platform compiler. Available for most operating systems.
Synapse and synaser are compatible with freepascal for windows and freepascal for linux.

It has numerous extensions for Delphi language features, c-style-operators (+:= *:=, etc.), multi-threaded and event-driven programming, that I know of. Even exceptions are supported.

You need to make sure that you enable the appropriate options, such as delphi 1 and 2 options, c style operators (sillyish). Just make sure to enable delphi 2 for the exceptions and classes.
Documentation is reasonable, but in program help is not as nice as the downloadable.

http://www.freepascal.org

HTTP

HyperText? Transfer Protocol

A protocol to transport (HTML)? documents, images and binary files.

Synapse unit : HTTPSend.pas
Synapse class : THTTPSend

rfc:
[rfc 1945] http 1.0
[rfc2616] http 1.1

links:
[w3c] your guide on HTML formated and related issues

OpenSource

Open Source is a way of releasing computer applications and libraries. Open source software and libraries can be released under several licenses.

For example, the GPL and LGPL licensing conditions forces everybody that uses (parts of) the code to release their software under GPL or LGPL conditions as well. For an overview and comparison of (L)GPL, please go to http://www.gnu.org

The BSD and MPL licenses are less restrictive. They allow the use of libraries and code in closed-source applications. This gives the programmer more freedom and the choice wether or not to release his or her software as open source.

Personally i flavour the BSD or MPL licenses, since there are cases where you do not want to disclose your application, but still want to use open source libraries.

Both Synapse and VisualSynapse are released under a BSD compatible license.

Open source sometimes is called 'CopyLeft?'. This may be confusing, since copyright will remain at the author. But the author clearly states you may use, modify and re-distribute the code.

VirtualPascal

Pascal compiler for OS/2 and win32.

http://www.vpascal.com/

Compatability with synapse: unknown

Windows

With the coming of Windows 3.1 microsoft had to deal with the multi-threading vs bhlocking sockets problem.
Before, on multitasking unix systems, sockets were mostly blocking.
Microsoft invented the (Asynchronous Socket)?, an event-orientated socket approach which would let single-threaded applications handle concurrent connections.
Synapse does not use asynchronous socket support by default. Blocking sockets commonly make protocol implementations easier, and are therefore favourized by many programmers.

Appendix B. Unsorted

Appendix C. Specifications

Features

Features of Synapse

Supported network- protocol related features:

- Full IPv6 support
- (SSL)? Support
- (Socks)? Support

Supported client protocols:

- HTTP
- (FTP)?
- (Telnet)?
- POP3
- (NNTP)?
- (IMAP)?
- SMTP
- (DNS)?
- (SNTP)?
- (LDAP)?
- (TFTP)?
- (Syslog)?
- (SNMP)?
- (Ping)?

Misc support

- Full (Mime)? implemtation
- Support functions, like
    - Encoding and decoding functions
        - (Base64)? Encode and Decode
        - (Quoted Printable)?
    - Conversion functions
        - MD5

This is list of basic features, as extracted from the official home page:

Synapse is not components, but only classes and routines. Not needed any installation! Just add unit to your uses.
Working under Windows and under Linux.
Can compile by Delphi, C++Builder, Kylix and FreePascal.
Support for communicating by dynamily loaded Winsock or Libc in blocking mode (or any other compatible library).
Supported is TCP, UDP, ICMP and RAW protocols.

Limited support for non-blocking communication mode.
Can use IPv4 and IPv6 addresses.
Native full SOCKS5 proxy support for TCP and UDP protocols.
Native full SOCKS4/4a proxy support for TCP protocol.
Support for TCP through HTTP proxy tunnel.
Support for TCP with SSL/TLS by OpenSSL? or SSLeay.
Support for TCP with SSL/TLS by StreamSecII?.
Support for PING request by ICMP or ICMPv6.
Support for ASN.1 coding and decoding.
Support for DNS (TCP or UDP) with many non-standard records (include zone transfers).
Support for character code transcoding. Supported charsets are basic ISO codings (ISO-8859-x), windows codings (CP-125x), KOI8-R, CP-895 (Kamenicky), CP-852 (PC-Latin-2) and UNICODE (UCS-4, UCS-2, UTF-7 and UTF-8).
Support for character replacing during character set transforms. (i.e. for remove diakritics, etc.)
Support for coding and decoding MIME e-mail messages (include character conversion of all supported charsets), include inline MIME encoding.
Support for SMTP and ESMTP protocol. SSL/TLS mode also supported.
Support for HTTP protocol 0.9, 1.0 and 1.1. Can handle ANY HTTP method, KeepAlives?, 100-status, Cookies and partial document downloading. Https also supported.
Support for SNMP protocol (include traps). Easy getting SNMP tables, etc.
Support for NTP and SNTP protocol (include broadcast client).
Support for POP3 protocol (classic and APOP login). SSL/TLS mode also supported.
Support for FTP protocol (support many firewalls include customs, upload and dowload resumes, transfer between two FTP servers). Implemented directory list parsing too. SSL/TLS support.
Support for TFTP protocol (cliend and server).
Support for LDAP protocol.
Support BSD Syslog client for unified platform independent logging capability.
Support for NNTP (Network News Transfer Protocol) include SSL/TLS support.
Support for Telnet script client.
Support for Base64 and Quoted-pritable coding and decoding.
Support for UUcode, XXcode and Yenc decoding.
Support for calculating CRC16, CRC32, MD5 and HMAC-MD5.
Support for autodetecting DNS servers or proxy settings.
Wake-on-lan

Appendix D. Code examples

```
unit rawIP;
{ Version 0.1
 This unit is explitely designed for windows. Sending raw sockets may succeed on
linux.
 The sniffing socket is solely for windows 2000/XP and needs administrator
rights to run.
 Based on synapse blocking sockets are two decsendants:
 *TSniffingSocket for receiving all network packets
 *TRawUDPSocket example of sending raw IP headers, this implementation can spoof
the sender.
 //FRemoteSin as declared in blcksock.pas must be moved from private to
 //protected in order for TRawUDPSocket unit to use it.
 Also, there is a
 *TSniffer object, that encapsulates a sniffer thread.
 for system information from ip helper api, there is:
 *TIPHelper
 see docs on synapse.wiki.dubaron.com for how to use.
 licensed under synapse compatible BSD license
 Author:
 Rene Tegel rene@dubaron.com
 release 0.1 augustus 17-19 2003.
}
interface
uses Windows, classes, Sysutils, {winsock,}synautil,
 blcksock, synsock, dialogs;
{
Usefull links:
Sending raw sockets:
http://msdn.microsoft.com/library/en-us/winsock/winsock/tcp_ip_raw_sockets_2.asp
Socket options:
http://msdn.microsoft.com/library/en-
us/winsock/winsock/tcp_ip_socket_options_2.asp
changes between wsock32.dll and ws2_32.dll:
http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/s
upport/kb/articles/Q257/4/60.ASP&NoWebContent=1
sniffing:
http://msdn.microsoft.com/library/default.asp?url=/library/en-
us/winsock/winsock/wsaioctl_2.asp
http://www.securityfriday.com/Topics/caputure_packet.html
}
{relevant RFC's:
 IP: RFC 791 IP
 950 subnetting
 919 broadcasting
 922 broadcastinf and subnets
 792 ICMP
 1112 Multicasting
 UDP: RFC 768
 TCP: RFC 793
 IP, ARP and FDDI: RFC 1390
 ARP and Ethernet: RFC 826
 IP and Ethernet: RFC 894
 IP and IEEE 802: RFC 1042
 PPP: RFC 1661 Point-to-Point protocol
 RFC 1662 PPP and HDLC framing
}
```

```
//winsock 2 declarations
const SIO_RCVALL = $98000001;
//raw IP packet related constants:
const //Ethernet header flags:
 EthernetType2 = $0800;
 EthernetTypeARP = $0806;
 //IP header flages
 FlagUnused = 1 shl 15;
 FlagDontFragment = 1 shl 14;
 FlagMore = 1 shl 13;
 //ICMP predefined protocol constants:
 ICMP_Echo_Reply = 0;
 ICMP_Dest_Unreachable = 3;
 ICMP_Source_Quench = 4;
 ICMP_Redirect = 5;
 ICMP_Echo_Request = 8;
 ICMP_Router_Advertisement=9;
 ICMP_Router_Sollicitation = 10;
 ICMP_Time_Exceeded = 11;
 ICMP_Parameter_Problem = 12;
 ICMP_Timestamp_Request = 13;
 ICMP_Timestamp_Reply = 14;
 ICMP_Information_Request = 15;
 ICMP_Information_Reply = 16;
 ICMP_Address_Mask_Request = 17;
 ICMP_Address_Mask_Reply = 18;
 ICMP_TraceRoute = 30;
 //TCP header flags:
 TCP_FIN = 1; //Connection control
 TCP_SYN = 2; //Synchronize, connection control, syncing sequence numbers
 TCP_RST = 4; //RESET, end of connection
 TCP_PSH = 8; //PUSH
 TCP_ACK = 16; //Acknowledgement number is valid
 TCP_URG = 32; //Urgent pointer is valid
// IP / TCP / UDP / ICMP / ARP header structures:
//only IP and UDP headers are tested.
//for other structures i cannot guarantee they are correct.
type Byte6 = array[0..5] of byte;
 Byte3 = array[0..2] of byte;
 TIPHeader = packed record
 case Integer of
 0: (
 VerLen: Byte;
 TOS: Byte;
 TotalLen: Word;
 Identifer: Word;
 FragOffsets: Word;
 TTL: Byte;
 Protocol: Byte;
 CheckSum: Word;
 SourceIp: DWORD;
 DestIp: DWORD;
 // Options: DWORD; //no options by default, header size 5 DWords
 );
 1: (Raw: Array[0..9] of Word);
 end;
 PIPHeader = ^TIPHeader;
 TArpHeader = packed record
 //Ethernet type typically $0806
 Hardware: Word;
 Protocol: Word;
 HardwareAddressLength:byte;
 ProtocolLength: byte;
 Operation:Word;
```

```
 SenderHWAddress:Byte6;
 SenderIPAddress:DWord;
 TargetHWAddress:Byte6;
 TargetIPAddress:DWord;
 end;
 TEthernetHeader = packed record
 DestMacAddress:Byte6; //leave $FF FFFF for broadcasts
 SourceMacAddress:Byte6;
 ProtocolType:Word; //mostly is EthernetType2
 end;
{ An EthernetPacket typically looks as follows:
 * TEthernetHeader
 * TIPHeader
 * TCP or UDP header
 * User data (TCP/UDP)
 * CRC checksum, DWord. wrapped by the ethernet protocol.
 so there is no real use for checksumming inside the user data.
}
 TICMPHeader = packed record
 ICMPtype:byte;
 code:byte;
 Checksum:Word;
 end;
 TICMPPacket = packed record //??
 IPHeader:TIPHeader;
 ICMPHeader:TICMPHeader;
 Data:Array[0..1499] of byte;
 end;
 TTCPHeader = packed record
 SourcePort:Word;
 DestPort:Word;
 SequenceNumber:DWord;
 AcknowledgementNumber:DWord;
 Offset:Byte; //only left 4 bits. Header length in 32-bit segments
 Flags:Byte;
 Window:Word;
 Checksum:Word; //includes speudo header instead of TCP header.
 UrgentPointer:Word;
 {Optionally:
 Options:byte3; //MSS (Maximum Segment Size) at connection startup
 Padding:byte;
 }
 end;
 TTCPPacket = packed record
 IPHeader:TIPHeader;
 TCPHeader:TTCPHeader;
 Data:Array[0..32767] of byte;
 end;
 TUDPHeader = packed record
 case Integer of
 0: (
 SourcePort,
 DestPort : Word; //why why why a Dword ???
 Length,
 Checksum:word;
 );
 1: (
 Raw:Array[0..3] of Word;
 );
 end;
 PUDPHeader = ^TUDPHeader;
// Helper functions from the ip helper api (iphlpapi.dll)
// Next types extracted from whirwater:
//
```

http://www.whirlwater.com/frames.php?http://www.whirlwater.com/information/2001/windows2000/usingtheiphelperapi.html

```
// thanx for coverting type definitions fellows
const
 MAX_HOSTNAME_LEN = 128; { from IPTYPES.H }
 MAX_DOMAIN_NAME_LEN = 128;
 MAX_SCOPE_ID_LEN = 256;
 MAX_ADAPTER_NAME_LENGTH = 256;
 MAX_ADAPTER_DESCRIPTION_LENGTH = 128;
 MAX_ADAPTER_ADDRESS_LENGTH = 8;
type
 TIPAddressString = Array[0..4*4-1] of Char;
 PIPAddrString = ^TIPAddrString;
 TIPAddrString = Record
 Next : PIPAddrString;
 IPAddress : TIPAddressString;
 IPMask : TIPAddressString;
 Context : Integer;
 End;
 PFixedInfo = ^TFixedInfo;
 TFixedInfo = Record { FIXED_INFO }
 case integer of
 0: (
 HostName : Array[0..MAX_HOSTNAME_LEN+3] of Char;
 DomainName : Array[0..MAX_DOMAIN_NAME_LEN+3] of Char;
 CurrentDNSServer : PIPAddrString;
 DNSServerList : TIPAddrString;
 NodeType : Integer;
 ScopeId : Array[0..MAX_SCOPE_ID_LEN+3] of Char;
 EnableRouting : Integer;
 EnableProxy : Integer;
 EnableDNS : Integer;
 );
 1: (A:Array[0..2047] of byte);
 End;
 PIPAdapterInfo = ^TIPAdapterInfo;
 TIPAdapterInfo = Record { IP_ADAPTER_INFO }
 Next : PIPAdapterInfo;
 ComboIndex : Integer;
 AdapterName : Array[0..MAX_ADAPTER_NAME_LENGTH+3] of Char;
 Description : Array[0..MAX_ADAPTER_DESCRIPTION_LENGTH+3] of Char;
 AddressLength : Integer;
 Address : Array[1..MAX_ADAPTER_ADDRESS_LENGTH] of Byte;
 Index : Integer;
 _Type : Integer;
 DHCPEnabled : Integer;
 CurrentIPAddress : PIPAddrString;
 IPAddressList : TIPAddrString;
 GatewayList : TIPAddrString;
 DHCPServer : TIPAddrString;
 HaveWINS : Bool;
 PrimaryWINSServer : TIPAddrString;
 SecondaryWINSServer : TIPAddrString;
 LeaseObtained : Integer;
 LeaseExpires : Integer;
 End;
//now some examples of support functions.
//maybo todo: create IPHelper object from this.
//get primary IP of each adapter on the system:
function GetAdapters(Strings:TStrings):Boolean;
//list all available IP's of this system:
//function getLocalIPList(Strings:TStrings):Boolean;
//convert local adapter IP address to mac address:
//function GetMacAddress (Adapter:String):String;
```

```
//lists actual systems dns servers:
//function ListDNSservers(Strings:TStrings):Boolean;
//externals:
function GetNetworkParams(FI : PFixedInfo; Var BufLen : Integer) : Integer;
 StdCall; External 'iphlpapi.dll' Name 'GetNetworkParams';
function GetAdaptersInfo(AI : PIPAdapterInfo; Var BufLen : Integer) : Integer;
 StdCall; External 'iphlpapi.dll' Name 'GetAdaptersInfo'
////////////////////////////////////////////////////////////////
 //Classes that implement protocols on top of raw IP:
type
 TRawUDPBlockSocket = class (TBlockSocket)
 public
 IPHeader:TIPHeader;
 UDPHeader:TUDPHeader;
 Data:Array[0..2047] of byte;
 procedure CreateSocket;// override;
 procedure CalcUDPChecksum;
 procedure Connect(IP, Port: string); override;
 procedure SetFrom(IP, Port: string);
 function SendBuffer(Buffer: Pointer; Length: Integer): Integer; override;
 end;
 TSniffingSocket = class (TBlockSocket)
 FAdapterIP:String;
 procedure CreateSocket;
 end;
 TOnPacketSniffed = procedure (Sender:TObject; Data:String) of Object;
 TSnifferThread = class (TThread)
 FData:String;
 FOwner:TObject;
 FSocket:TSniffingSocket;
 procedure SyncPacket;
 procedure Execute; override;
 end;
 TSniffer = class (TObject) //make it component if you like
 private
 FOnPacket:TOnPacketSniffed;
 protected
 FActive:Boolean;
 public
 FSniffer:TSnifferThread;
 FAdapter:String;
 procedure SetActive(Value:Boolean);
 constructor Create;
 destructor Destroy;
 published
 property Adapter:String read FAdapter write FAdapter;
 property Active:Boolean read FActive write SetActive;
 property OnPacketSniffed:TOnPacketSniffed read FOnPacket write FOnPacket;
 end;
 //ip helper interface
 TIPHelper = class (TObject) //make that component if you like
 //After construction, these strings will be created and filled
 //system wide settings:
 HostName : String;
 DomainName : String;
 CurrentDNSServer : String;
 DNSServerList : TStrings;
 NodeType : Integer;
 ScopeId : String;
 EnableRouting : Boolean;
 EnableProxy : Boolean;
 EnableDNS : Boolean;
 //Filled per adapter:
 DNSServers:TStrings;
```

```
 AdapterIPs:TStrings;
 AdapterNames:TStrings;
 AdapterDescriptions:TStrings;
 AdapterMACs:TStrings;
 DHCPServers:TStrings;
 GateWays:TStrings;
 CurrentIPs:TStrings;
 CurrentMasks:TStrings;
// PrimaryIPs:TStrings;
// PrimaryMasks:TStrings;
 //LeaseObtained:TList
 //LeaseExpired:TList
 //multiples filled per adapter
 AllIPS:TStrings;
 AllMasks:TStrings;
 constructor Create;
 destructor Destroy;
 end;
implementation
//support functions:
constructor TIPHelper.Create;
var Data:String;
 l:Integer;
 PInfo:PIPAdapterInfo;
 PIP : PIPAddrString;
 NWInfo:PFixedInfo;
 M:String;
 i:Integer;
 procedure AddrToStrings (P:PIPAddrString; IP:TStrings; Mask:TStrings);
 begin
 while P<>nil do
 begin
 if Assigned (IP) then IP.Add(P^.IPAddress);
 if Assigned (Mask) then Mask.Add(P^.IPMask);
 P := P^.next;
 end;
 end;
begin
 inherited;
 DNSServerList:=TStringList.Create;
 DNSServers:=TStringList.Create;
 AdapterIPs:=TStringList.Create;
 AdapterNames:=TStringList.Create;
 AdapterDescriptions:=TStringList.Create;
 AdapterMACs:=TStringList.Create;
 DHCPServers:=TStringList.Create;
 GateWays:=TStringList.Create;
 CurrentIPs:=TStringList.Create;
 CurrentMasks:=TStringList.Create;
// PrimaryIPs:=TStringList.Create;
// PrimaryMasks:=TStringList.Create;
 //LeaseObtained:TList
 //LeaseExpired:TList
 //multiples filled per adapter
 AllIPS:=TStringList.Create;
 AllMasks:=TStringList.Create;
 //Now fill structures
 //Fill Strings with an array of adapters
 SetLength (Data, 8192); //arbritrary, increase if you expect loads of adapters.
 PInfo := @Data[1];
 l:=length(Data);
 if 0 = GetAdaptersInfo (PInfo, l) then
 //now PInfo contains list of adapters:
 while (PInfo<>nil) and
```

```
 (Integer(PInfo)<=Integer(@Data[Length(Data)])-SizeOf(TIPAdapterInfo)) do
 begin
 AdapterNames.Add (PInfo^.AdapterName);
 AdapterDescriptions.Add (PInfo^.Description);
 M:='';
 for i:= 1 to PInfo^.AddressLength do
 M:=M+IntToHex (byte(PInfo^.Address[i]), 2);
 AdapterMacs.Add (M);
 if Assigned (PInfo^.CurrentIPAddress) then
 begin
 CurrentIPs.Add(String(PInfo^.CurrentIPAddress^.IPAddress));
 CurrentMasks.Add(PInfo^.CurrentIPAddress^.IPMask);
 end;
 AddrToStrings (@PInfo^.GatewayList, GateWays, nil);
 AddrToStrings (@PInfo^.DHCPServer, DHCPServers, nil);
 AddrToStrings (@PInfo^.IPAddressList, AllIPs, AllMasks);
 PInfo := PInfo^.Next;
 end;
 //Now fill system-wide settigs:
 NWInfo := @Data[1];
 if 0=GetNetworkParams(NWInfo, l) then
 begin
 Hostname := NWInfo^.HostName;
 DomainName := NWInfo^.DomainName;
 if Assigned (NWInfo^.CurrentDNSServer) then
 CurrentDNSServer := NWInfo^.CurrentDNSServer^.IPAddress;
 AddrToStrings (@NWINfo^.DNSServerList, DNSServers, nil);
 EnableRouting := boolean (NWInfo^.EnableRouting);
 EnableProxy := boolean (NWInfo^.EnableProxy);
 EnableDNS := boolean(NWInfo^.EnableDNS);
 ScopeID := NWInfo^.ScopeId;
 NodeType := NWInfo^.NodeType;
 end;
end;
destructor TIPHelper.Destroy;
begin
 DNSServerList.Free;
 DNSServers.Free;
 AdapterIPs.Free;
 AdapterNames.Free;
 AdapterDescriptions.Free;
 AdapterMACs.Free;
 DHCPServers.Free;
 GateWays.Free;
 CurrentIPs.Free;
 CurrentMasks.Free;
// PrimaryIPs.Free;
// PrimaryMasks.Free;
 //LeaseObtained.Free
 //LeaseExpired.Free
 AllIPS.Free;
 AllMasks.Free;
 inherited;
end;
function getAdapters;
var Data:String;
 l:Integer;
 PInfo:PIPAdapterInfo;
 PIP : PIPAddrString;
begin
 //Fill Strings with an array of adapters
 Result := False;
 if (Strings=nil) or not (Strings is TStrings) then
 exit;
```

```
 Strings.Clear;
 SetLength (Data, 8192); //arbritrary, increase if you expect loads of adapters.
 PInfo := @Data[1];
 l:=length(Data);
 if 0 <> GetAdaptersInfo (PInfo, l) then
 exit;
 //now PInfo contains list of adapters:
 while (PInfo<>nil) and
 (Integer(PInfo)<=Integer(@Data[Length(Data)])-SizeOf(TIPAdapterInfo)) do
 begin
 PIP := @PInfo^.IPAddressList;
 while PIP<>nil do
 begin
 Strings.Add (PIP^.IPAddress);
 PIP := PIP^.Next;
 Result := True;
 end;
 PInfo := PInfo^.Next;
 end;
end;
procedure TRawUDPBlockSocket.CreateSocket;
var c:Integer;
 Sin: TVarSin;
 IP:String;
 Port:String;
begin
 FSocket := synsock.Socket(AF_INET, SOCK_RAW, IPPROTO_IP);
 inherited CreateSocket;
 c:=1;
 setsockopt(FSocket, 0{SOL_SOCKET}, IP_HDRINCL, @c, sizeof(c));
 //fill header info
 with IPHeader do
 begin
 Protocol := 17; //UDP
 TTL := 128;
 VerLen := (4 shl 4) or 5;
 end;
end;
procedure TRawUDPBlockSocket.CalcUDPChecksum;
//calc UDP checksum
var checksum:Integer;
 i,l,m:Integer;
begin
 // see rfc 768;
 // http://www.faqs.org/rfcs/rfc768.html
 checksum := 0;
 l := ntohs(UDPHeader.Length) - SizeOf(TUDPHeader); //data length
 m := l div 2; //see if padding is needed
 if (l mod 2)<>0 then
 begin
 Data[l] := 0; //add padding zero
 inc (m);
 end;
 //checksum the data:
 for i:=0 to m-1 do
 Checksum := Checksum - ((Data[i*2] shl 8) + Data[i*2+1])-2;
 //pseudo headers, source+dest:
 for i:= 8 to 9 do
 Checksum := Checksum - IPHeader.Raw[i] -1;
 //pseudo headers: proto + udplength
 Checksum := Checksum - IPHeader.Protocol - UDPHeader.Length -2;
 //take the one's complement:
 Checksum := - Checksum - 1;
 //now make 16 bits from 32 bits:
```

```
 while (Checksum shr 16)>0 do
 Checksum := ((Checksum shr 16) and $FFFF) or (Checksum and $FFFF);
 UDPHeader.Checksum := 0; //Checksum; it aint working yet.
end;
procedure TRawUDPBlockSocket.Connect;
type
 pu_long = ^u_long;
var HostEnt:PHostEnt;
begin
 //inherited connect is of no use here, since we deal with raw sockets.
 //fill the IP header structure
 inherited Connect (IP, Port); //fill sins
 if IP = cBroadCast then
 IPHeader.DestIP := INADDR_BROADCAST
 else
 begin
 IPHeader.DestIP := synsock.inet_addr(PChar(IP));
 if IPHeader.DestIP = INADDR_NONE then
 begin
 HostEnt := synsock.GetHostByName(PChar(IP));
 if HostEnt <> nil then
 IPHeader.DestIP := u_long(Pu_long(HostEnt^.h_addr_list^)^);
 end;
 end;
 UDPHeader.DestPort := htons (StrToIntDef (Port, 0));
end;
procedure TRawUDPBlockSocket.SetFrom;
type pu_long = ^ulong;
var HostEnt:PHostEnt;
begin
 if IP = cBroadCast then
 IPHeader.SourceIP := INADDR_BROADCAST
 else
 begin
 IPHeader.SourceIP := synsock.inet_addr(PChar(IP));
 if IPHeader.SourceIP = INADDR_NONE then
 begin
 HostEnt := synsock.GetHostByName(PChar(IP));
 if HostEnt <> nil then
 IPHeader.SourceIP := u_long(Pu_long(HostEnt^.h_addr_list^)^);
 end;
 end;
 UDPHeader.SourcePort := htons(StrToIntDef (Port, 0));
end;
function TRawUDPBlockSocket.SendBuffer;
var P:String; //the actual packet
 d:TSockAddr;
 l:Integer;
begin
 if Length>=high(Data) then
 begin
 Result := -1;
 exit;
 end;
 //set header checksum
 IPHeader.TotalLen := htons(SizeOf(TIPHeader)+SizeOf(TUDPHeader)+Length);
 IPHeader.Identifer := gettickcount mod $FFFF; //0 also allowed, then kernel
fills it.
 IPHeader.FragOffsets := $00; //htons(FlagDontFragment);
// CalcHeaderChecksum; //don't, kernel does this!
 //move data
 move (Buffer^, Data[0], Length);
 //set udp checksum
 UDPHeader.Length := htons(SizeOf(TUDPHeader)+Length);
```

```
 CalcUDPChecksum; //you can leave it zero if you like: UDPHeader.Checksum := 0;
 //move to buffer,
 //setlength of total IP packet:
 SetLength (P, SizeOf(TIPHeader)+SizeOf(TUDPHeader)+Length);
 //move IP header:
 move (IPHeader.Raw[0], Pointer(P)^, SizeOf (TIPHeader));
 //move IP data, in this case: UDP header:
 move (UDPHeader.Raw[0], P[1+SizeOf(TIPHeader)], SizeOf (TUDPHeader));
 //move UDP data:
 move (Data[0], P[1+SizeOf(TIPHeader)+SizeOf(TUDPHeader)], Length);
 //send data
 l:=system.Length(P);
// Connect (IPHeader.DestIP, IPHeader.Port);
 Result :=
 sendto (FSocket,
 P[1],
 l,
 0, //flags
 PSockAddr(@FRemoteSin), //we need this filled for the kernel to send.
 SizeOf(FRemoteSin));
end;
procedure TSniffingSocket.CreateSocket;
var c,d,l:Integer;
 F:TStrings;
 Sin: TVarSin;
begin
 FSocket := synsock.Socket(AF_INET, SOCK_RAW, IPPROTO_IP);
// no inherited CreateSocket here.
 c:=1;
 if FAdapterIP = '' then
 begin
 //fetch adapterIP
 //get default (first listed) adapter:
 F:=TStringList.Create;
 if (not GetAdapters(F)) or (F.Count=0) then
 exit; //don't try further, no use without IP.
 FAdapterIP := F[0];
 end;
 SetSin(Sin, FAdapterIP, '0');
 SockCheck(synsock.Bind(FSocket, @Sin, SizeOfVarSin(Sin)));
 c := 1;
 setsockopt(FSocket, 0{SOL_SOCKET}, IP_HDRINCL, @c, sizeof(c)); //not necessary
 c := 500000;
 setsockopt(FSocket, SOL_SOCKET, SO_RCVBUF, @c, sizeof(c)); //not necessary
 c:=1;
 d:=0;
 FLastError := WSAIoctl (FSocket, SIO_RCVALL, @c, SizeOf(c), @d, SizeOf(d),@l ,
nil, nil);
end;
procedure TSnifferThread.SyncPacket;
begin
 try
 if Assigned (TSniffer(FOwner).FOnPacket) then
 TSniffer(FOwner).FOnPacket (FOwner, FData);
 except //build-in safety, stop sniffing if anything fails:
 FOwner := nil;
 Terminate;
 end;
end;
procedure TSnifferThread.Execute;
begin
 FSocket.CreateSocket;
 while not Terminated do
 begin
```

```
 if (FSocket.WaitingData>0) and (FSocket.WaitingData<1000000) then
 begin
 SetLength (FData, FSocket.WaitingData);
 SetLength (FData,
 FSocket.RecvBuffer (@FData[1], length(FData)) );
 if FData<>'' then
 Synchronize (SyncPacket);
 end
 else
 sleep(0);
 end;
end;
constructor TSniffer.Create;
begin
 inherited;
end;
destructor TSniffer.Destroy;
begin
 Active := False;
 inherited;
end;
procedure TSniffer.SetActive;
begin
 if Value = FActive then
 exit;
 if Value then
 begin
 FSniffer := TSnifferThread.Create (True);
 FSniffer.FSocket := TSniffingSocket.Create;
 FSniffer.FSocket.FAdapterIP := FAdapter;
 FSniffer.FOwner := Self;
 FSniffer.FreeOnTerminate := True;
 FSniffer.Resume;
 end
 else
 begin
 FSniffer.Terminate;
 FSniffer.WaitFor;
 FSniffer.Free;
 end;
 FActive := Value;
end;
end.
{
procedure TRawBlockSocket.CalcHeaderChecksum;
//funny, we don't need it. the kernel does.
var i:Integer;
 checksum:DWord;
begin
 //calc the IP header checksum:
 IPHeader.CheckSum := htons(0); //fill with network zero, since it gets included
in packet as well.
 checksum := 0;
 for i:=0 to 9 do
 checksum := checksum + IPHeader.Raw[i];
 //shift and add high bytes:
 checksum := (checksum shr 16) + (checksum and $FFFF);
 //and add optional carry bit:
 inc (checksum, checksum shr 16);
// checksum := 1+ (not checksum); //complement
 checksum:=DWord(-Integer(checksum));
 IPHeader.CheckSum := htons(Checksum and $FFFF);
end;
}
```

udp server

How can I write UDP server?

Try next code on thread, or in any other case:

```
procedure TSomeThread.Execute;
var
 Sock:TUDPBlockSocket;
 size:integer;
 buf:string;
begin
 Sock:=TUDPBlockSocket.Create;
 try
 sock.bind('0.0.0.0','port');
 if sock.LastError<>0 then exit;
 while True do
 begin
 if terminated then break;
 buf := sock.RecvPacket(1000);
 if sock.lasterror=0 then
 begin
 // do something with data and prepare response data
 sock.SendString(Buf);
 end;
 sleep(1);
 end;
 sock.CloseSocket;
 finally
 sock.free;
 end;
end;
```

Appendix E. Articles from LukasGebauer

Binary strings

Binary strings

Long strings can be used for storing of any binary datas! Length of this string is stored outside of string. You can store to this string any byte value, include char #0!

This long strings is ideal for binary buffers, because it is automaticly allocated and deallocated from memory. By using of long string as binary buffers you can create memory-leak-free applications!

However graphical system using null-terminated string! You must be carefull with writing your binary strings to display! All labels or other windows control cannot displat this binary strings! It is true for debug tooltips too!

You must be careful with typecasting too. Any assigning to pchar can truncate this binary string too!

However when you use only long strings, then it is safe and useful!

C++Builder

How to use Synapse .pas files in Borland C++Builder's projects?

(respond by "Ozzman" <ozzman@softdepia.com>)

1. Needed to generate header files (hpp) for including in project (example for creatinh header file for dnssend.pas): DCC32.EXE -JPHW DNSSEND.PAS

2. Need to define path of .hpp(same directory with .pas) files:
Project->Add->Directories/Conditionals add in "Include path", "Library path" and "Debug source path" next string: "$(BCB)\synapse\source\lib"

3. Define path to .pas files in BCB enviroment variables:
Tools -> Environment Options ->Library "Library path" and "Browsing path".

including library to project:
Project -> Add to project -> DNSSEND.PAS (for example)

including header file:
unit1.h

---

1. include "DNSSend.hpp"

---

unit1.cpp

---

1. pragma link "DNSsend"

Possible, error can appear in line 180 of file blcksock.hpp:
__property WSAData WSAData = {read=FWsaData?};
I' just comment this line cause i don't need to use this variable:
// __property WSAData WSAData = {read=FWsaData?};

CreateSocket

CreateSocket issues

On very old Synapse versions you must allways call CreateSocket explicitly.

Then I do 'autocreate' socket capability. When You call Connect or Bind on non-created socket, then Synapse call CreateSocket. With this case you not need call CreateSocket explicitly. You need call this only on special cases.

From release 31 is CreateSocket changed!

By default, you not need call CreateSocket. It is created automaticly. When you have family on SF_Any (default), then Synapse must know IP address for determinig, if must cretae IPv4 or IPv6 socket. This is reason, why I moved creating of socket after name resolution in Connect or Bind method.

If you call Connect or Bind on non-created socket, then Synapse try to resolve given address, and create socket by family of given address.

When you call explicitly CreateSocket, then you break this autodetection. Never mind! When you set Family on SF_Any, then CreateSocket do nothing. When you set some socket properties, then it is remembered and aplied after real socket is created. It allows maximum possible compatibility with older code.

When you set family to IPv4 or Ipv6, then CreateSocket working on standard way.

Exists new function CreateSocketByName?. It have one argument for name. This function resolve given name first, and then create socket with family by given name.

When you have special case and need create socket before Connect or Bind, then you must set another family then SF_Any, or you must call CreateSocketByName?.

When you need only set special properties after socket creation and before using of Socks in Connect or Bind call, then you can use hook OnCreateSocket?.

IPv6

IPv6 issues

New version of Synapse with IPv6 support is transparent and allows instant using of IPv6 on commonly used code.

When you are using code without special code, then you simply recompile your program with new Synapse, and your code is IPv6 enabled! You can use IPv4 IPs, IPv6 IPs or IPv4 and IPv6 domain names. Synapse create socket itself by result of resolving addresses.

Becasue commonly used command 'Bind('0.0.0.0','0')' before Connect breaks socket type autodetection, this command do nothing, when SF_Any is used. In this case is used system implicit bind with same effect. You not need remove this call.

When you call Bind with another argument, then bind is called. This bind cause socket creation! When you bind socket to specific interface, then socket family usage is limited by binded IP family!

Not all systems support full IPv6. Look to other articles!

In some cases is automatic created socket not ideal. Look to article about CreateSocket.

IPv6_OS

What systems is supported for IPv6?

On linux:
You need kernel with IPv6 support. (all newer kernels 2.4.x) However usualy is Ipv6 support compiled as module. Try call 'modprobe ipv6' command for load IPv6 support.

On Windows:
Microsoft have official supported IPv6 implementation on WinXP? SP1 and newer operating systems.

Systems prior this version sometimes have 'prelimitary support'. This versions is for developer testing. It not have produce quality. Not all needed functions is supported in this versions!

Synapse supporting this prelimitary versions too, however autodetecting of socket type not working propery in all cases. Better way is set socket Family in your source to SF_IP4 or SF_IP6.

read data

How to read data by Synapse?

Not exists event for reading data in Synpase. Why? It is not needed! In other libraries you must read data when it income to your system by event. This event is outside from your processing procedure! Then you must parse this data and get what you need! You must synchronise your procedure with this events. It is not practical in most cases!

Synapse is synchronous library, because you not need do this synchronisation. All data read is synchronised with your processing code! You must not know when data arraive to your system. This datas is silently stored in internal buffers. You call read operation when you need read some data. Look, you call read operation when YOU need data, not when system need read data!

When you need read data, then you need:

- read known count of bytes
- read byte with know terminator
- read any data
- combination of above

1. read known count of bytes
You know how much bytes you need. You can use RecvBuffEx? method for receive specific numbr of bytes to your binary buffer. You can use RecvBuffStr? too. This function read bytes to binary string. This string is good as dynamicly allocated binary buffer. (see to other article)

2. read byte with know terminator
You know sequence what terminating your data block. Then you can use RecvTerminated? method. Special case is reading lines of text terminated by CRLF. For this you can use special RecvString? method. When you use this method and turn on ConvertLineEnd? property, then Synapse autodetect correct line terminating sequence. It can receive lines terminated by CRLF, LFCR, CR or LF! It is very good for your bug-tolerant application.

3. read any data
You simply need receive any data, regardless count. You need fetch any incomming data. Then you can use RecvPacket? method.

4. combination of above
You can freely combine all previous method. You can read one line with length (in text form) of next data block and then you can read this data block by RecvBuffStr?, for example.

Stringlists

Huge using of Stringlists causing huge memory allocations.

The easier way to fix the problem is to let Windows handle all the memory
for you since it doesn't keep a heap full of holes in allocated memory.
Here's a very simple alternate memory manager unit. Just insert it as the
first unit in the uses statement in the .DPR file and the problem goes away.

```
[MemoryManager.pas]
unit MemoryManager;
interface
uses SysUtils;
type EMemoryManager = class(Exception);
implementation
uses Windows;
function MGetMem(Size : Integer) : Pointer;
begin
 if Size=0 then raise EMemoryManager.Create('Why did you pass MGetMem Size=0?
You know I can''t do that.');
 Result:=GlobalAllocPtr(GMEM_FIXED, Size);
 if Result=nil then raise EMemoryManager.CreateFmt('MGetMem failed while trying
to allocate %d bytes, error %d', [Size, GetLastError]);
end;
function MFreeMem(P : Pointer) : Integer;
begin
 Result:=GlobalFreePtr(P);
 if Result<>0 then raise EMemoryManager.CreateFmt('MFreeMem failed while trying
to free %p, error %d', [P, GetLastError]);
end;
function MReallocMem(P : Pointer; Size : Integer) : Pointer;
begin
 Result:=GlobalReallocPtr(P, Size, GMEM_MOVEABLE);
 if Result=nil then raise EMemoryManager.CreateFmt('MReallocMem failed while
trying to reallocate %p from %d to %d bytes, error %d', [P,
GlobalSize(HGLOBAL(P)), Size, GetLastError]);
end;
const NewMemoryManager : TMemoryManager = (GetMem: MGetMem; FreeMem: MFreeMem;
ReallocMem: MReallocMem);
initialization
 SetMemoryManager(NewMemoryManager);
end.
```

SyncObjs

Synapse using SyncObjs unit, but I not have them!

In some versions of Delphi this unit missing. (I don't know why...)

You can download it from "contributed download area".
[syncobjs.pas]

TMimeMess

Abstract

Synapse class implementing a mime-formatted message. Commonly used to encode/decode email messages. TMimeMess contains one or more TMimeParts?.
TMimeMess provides support for assembling, encoding/decoding and modifying mime messages.

Example

```
procedure TForm1.Button13Click(Sender: TObject);
var
 m:TMimemess;
 l:tstringlist;
 p: TMimepart;
begin
 m:=3DTMimemess.create;
 l:=3Dtstringlist.create;
 try
 p := m.AddPartMultipart('mixed', nil);
 l.loadfromfile('c:\search.log');
 m.AddPartText(l,p);
 m.AddPartBinary('c:\search.log',p);
 m.header.from:='"Your Name" <I@somewhere.com>';
 m.header.tolist.add('You@somewhere.com');
 m.header.subject:='pokusna zprava';
 m.EncodeMessage;
 memo1.lines.assign(m.lines);
 finally
 m.free;
 l.free;
 end;
end;
```

See MailAttach for another example.

Related subjects: TMimePart TSMTPSend TSendMail

TMimePart

Here is some word about new TMimePart:

For access to subpart is new methods:

- GetSubPartCount? return count of subpart. (it is like Count method in stringlists.)

- GetSubPart? return subpart with reqested index. You can use for cycle from 0 to GetSubPartCount?-1 for access to all subparts on this message level.

- Addsubpart add new subpart. ;-)

- ClearSubParts? clears and destroy all subparts.

This function allows easy access to any part in message.

example:

You have tmimepart object called 'm', where is decomposed message. You need get headers of first subpart, whitch is second subpart of root multipart. For this you can use:

m.getsubpart(1).getsubpart(0).Headers

UDP multicasts

How to use UDP multicasts?

This is few examples of using multicasts from SynaList?.
It is based on multicast demo from microsoft MSDN.

Description:
234.5.6.7:22401 is any class D IP address + port

```
procedure MulticastSendTest;
var
 sndsock:TUDPBlockSocket;
begin
 sndsock:=TUDPBlockSocket.Create;
 try
 sndsock.createsocket;
 sndsock.Bind('0.0.0.0','0');
 sndsock.MulticastTTL := 1;
 sndsock.connect('234.5.6.7','22401');
 sndsock.SendString('Ahoy!'+CRLF);
 finally
 sndsock.free;
 end;
end;
procedure MulticastRecvTest;
var
 rcvsock:TUDPBlockSocket;
 buf:string;
begin
 rcvsock:=TUDPBlockSocket.Create;
 try
 rcvsock.createsocket;
 rcvsock.Bind('0.0.0.0','22401');
 rcvsock.AddMulticast('234.5.6.7');
 buf:=rcvsock.RecvPacket(60000);
 showmessage(buf);
 finally
 rcvsock.free;
 end;
end;
...or this test in one procedure:
procedure multicasttest;
var
 sndsock:TUDPBlockSocket;
 rcvsock:TUDPBlockSocket;
 buf:string;
begin
 sndsock:=TUDPBlockSocket.Create;
 rcvsock:=TUDPBlockSocket.Create;
 try
 rcvsock.createsocket;
 rcvsock.Bind('0.0.0.0','22401');
 rcvsock.AddMulticast('234.5.6.7');
 if rcvsock.LastError <> 0 then exit; //JK: "<>" was missing
 sndsock.createsocket;
 sndsock.Bind('0.0.0.0','0');
 sndsock.MulticastTTL := 1;
 sndsock.connect('234.5.6.7','22401');
 if sndsock.LastError <> 0 then exit; //JK: ":=" always exits
 sndsock.SendString('Test Payload');
 buf:=rcvsock.RecvPacket(1000);
```

```
 showmessage(buf);
 sndsock.CloseSocket;
 rcvsock.CloseSocket;
 finally
 sndsock.free;
 rcvsock.free;
 end;
end;
```

UDP server

How can I write UDP server?

Try next code on thread, or in any other case:

```
procedure TSomeThread.Execute;
var
 Sock:TUDPBlockSocket;
 size:integer;
 buf:string;
begin
 Sock:=TUDPBlockSocket.Create;
 try
 sock.bind('0.0.0.0','port');
 if sock.LastError<>0 then exit;
 while True do
 begin
 if terminated then break;
 buf := sock.RecvPacket(1000);
 if sock.lasterror=0 then
 begin
 // do something with data and prepare response data
 sock.SendString(Buf);
 end;
 sleep(1);
 end;
 sock.CloseSocket;
 finally
 sock.free;
 end;
end;
```

Appendix F. Tutorial 1

Synapse in 2 hours

Synapse in two hours

- Lesson0 - Introduction
- Lesson1 - Using THTTPSend to retrieve a html
- Lesson2 - Creating a TTCPBlockSocket to connect to a server
- Lesson3 - Building threads

Lesson0

Remember that Synapse is Object oriented. So except for some support functions, we always need to create one.
Also, synapse libraries are of class TObject. So, they are not components. A project TVisualSynapse may get started soon.

The two most used classes of synapse are TTCPBlockSocket and TUDPBlockSocket. Also, there are classes for specific internet protocols. All protocol client implemenions have the name of the protocol and 'send' in it. THTTPSend, TPOP3Send, (TIMAPSend)?, TSMTPSend, (TDNSSend)?, (TLDAPSend)? etc.
These objects are wrappers around the TTCPBlockSocket class or the TUDPBlockSocket class. These objects are clients, not servers. Most server implementations you have to write yourself, but as you will see, it is not hard to listen on a port and communicate with clients.

Do me a real example: Lesson1

Lesson1

pascal units

Most of the time you want to include two or three, or more, synapse units: The protocol you are using, say TTCPBlockSocket or THTTPSend; blcksock.pas which defines some constants, and synautil.pas, which contains some support functions.

- uses Sysutils, HTTPSend, blcksock;

This example uses Delphi, other compilers may be simulair:

This program will head the http headers from the www.w3c.com default index page, and get the contents of the default page at www.microsoft.com.

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils, HTTPSend;
```

```
var H:THTTPSend;
 S:String;
begin
 H := THTTPSend.Create;
 H.HTTPMethod ('head', 'http://www.w3c.com/');
 writeln (H.Headers.Text);
 writeln ('This was the header as sent from the w3c server - press
enter');
 readln;
 //Don't forget to clear the headers with re-use, else they will be
sent:
 H.Headers.Clear;
 H.HTTPMethod ('get', 'http://www.microsoft.com/');
 SetLength (S, H.Document.Size);
 H.Document.Read (S[1], Length(S));
 writeln (S);
 writeln ('This was a GET request. Press enter');
 readln (S);
end.
```

>> Lesson2

Lesson2

This example will show how you could use TTCPBlockSocket:

```
program Project1;
{$APPTYPE CONSOLE}
uses SysUtils, BlckSock;
var H:TTCPBlockSocket;
 S:String;
begin
 H := TTCPBlockSocket.Create;
 H.Connect ('www.w3c.com', '80');
 begin
 H.SendString ('GET / HTTP/1.0'#13#10#13#10);
 while (H.LastError = 0) do
 begin
 S:=H.RecvString(5000);
 writeln (S);
 end;
 end;
 H.Free;
 writeln ('press enter');
 readln;
end.
```

This example uses SendString? and RecvString? methods. There are other, methods available to send and receive data, like sendbuffer and recvbuffer.

RecvString? reads a line that is terminated by the CarriageReturn? / LineFeed? (#13#10) combination. It will just wait until the line terminate sequence comes by. In this case it will time-out in 5 seconds.

SendString? on the contrary will *not* send the Carriage Return or the LineFeed?. You will have to add them yourself.

RecvString? is a very handy procedure if you are implementing some line-based protocol, like POP3 or IMAP.
It is of no use if you receive (binary data)?. You should use the RecvBuffer? method for that purpose.

Next: Synapse and threads

Lesson3

Synapse and Threads

The blocking socket behaviour of synapse almost force you to write threads, at least: if you want your application to be response. Except when you would write, for example, a very simple UDP server as service, you almost have to. And even then, it is a good idea to add a thread class, since you create a nice [[Object]].

So, a typical unit using using Synapse would contain a declaration like this:

```
unit MyUnit;
interface
uses Classes, BlckSock;
// optionally you probably use one of those units: Windows, Sysutils, SyncObjs,
SynaUtil
type
 TMyTCPClient = class (TThread)
 FSock:TTCPBlockSocket;
 procedure Execute; override;
 end;
```

In the implementation part you only have to fill in the threads execute code:

```
procedure TMyTCPClient.Execute;
begin
 FSock := TTCPBlockSocket.Create;
 FSock.Connect ('someserver', '1234'); //server, portnumber
 while not Terminated and (FSock.LastError = 0) do
 begin
 if length(SomeBuf)>0 then
 //Send data
 FSock.SendString(SomeBuf);
 if FSock.CanRead (200) then
 begin
 i:=FSock.WaitingData;
 if i>0 then
 begin
 SetLength (InBuf, i);
 FSock.RecvBuffer (@Inbuf[1], i);
 //Do some with Inbuf
 end;
 end;
 end;
 FSock.Free;
end;
```

Now, suppose we want to share this data with some other thread, like the main application.
We add SyncObjs to the uses clause and add this to the declaration of our thread:

```
type
 TOnTCPDataReceived = procedure (Sender: TObject; Data:String);
 TMyTCPClient = class (TThread)
 FSock:TTCPBlockSocket;
 //Shared data
 private
 FDataOut:String;
 FDataIn:String;
 CSData:TCriticalSecion;
 FOnData:TOnTCPDataReceived;
 procedure SyncDataReceived;
 public
 procedure SendData (Data:String);
 procedure Execute; override;
 property OnDataReceived: TOnTCPDataReceived read FOnData write FOnData;
 end;
```

We will implement produre SendData? as follows:

```
procedure TMyTCPClient.SendData;
begin
 CSData.Enter;
 try
 FDataIn := FDataIn + Data;
 finally
 CSData.Leave;
 end;
end;
```

and SyncDataReceived? as follows:

```
procedure TMyTCPClient.SyncDataReceived;
begin
 if Assinged (FOnData) then
 try //we do not have to use critical section here because the calling thread is
paused during synchronize
 FOnData (Self, BufOut);
 except
 FOnData := nil;
 end;
end;
```

and, we add this line somewhere in our thread.execute just after data is received:

```
 synchronize (SyncDataReceived);
```

Lets put it all together before going to Lesson4 :

```
unit myunit;
interface
uses {Windows, }Sysutils, Classes, BlckSock, SyncObjs, SynaUtil;
type
 TOnTCPDataReceived = procedure (Sender: TObject; Data:String);
 TMyTCPClient = class (TThread)
 FSock:TTCPBlockSocket;
 //Shared data
```

```
 private
 FDataOut:String;
 FDataIn:String;
 CSData:TCriticalSection;
 FOnData:TOnTCPDataReceived;
 procedure SyncDataReceived;
 public
 procedure SendData (Data:String);
 procedure Execute; override;
 property OnDataReceived: TOnTCPDataReceived read FOnData write FOnData;
 end;
implementation
procedure TMyTCPClient.Execute;
var S:String;
 i:Integer;
begin
 FSock := TTCPBlockSocket.Create;
 FSock.Connect ('someserver', '1234'); //server, portnumber
 while not Terminated and (FSock.LastError = 0) do
 begin
 CSData.Enter;
 S:=FDataIn;
 FDataIn := '';
 CSData.Leave;
 if length(S)>0 then
 //Send data
 FSock.SendString(S);
 if FSock.CanRead (200) then
 begin
 i:=FSock.WaitingData;
 if i>0 then
 begin
 SetLength (FDataIn, i);
 SetLength (FDataIn,
 FSock.RecvBuffer (@FDataIn[1], i)
 );
 //Do some with Inbuf
 synchronize (SyncDataReceived);
 end;
 end;
 end;
 FSock.Free;
end;
procedure TMyTCPClient.SendData;
begin
 CSData.Enter;
 try
 FDataIn := FDataIn + Data;
 finally
 CSData.Leave;
 end;
end;
procedure TMyTCPClient.SyncDataReceived;
begin
 if Assigned (FOnData) then
 try //we do not have to use critical section here because the calling thread is
paused during synchronize
 FOnData (Self, FDataIn);
 except
 FOnData := nil;
 end;
end;
end.
```

Lesson4 >>

Lesson4

The lessons split up here. I invite you too take a look at LessonProxy or SimpleTunnel? or for examples of server threads.

In LessonProxy i show and discuss an example of a proxy server.

In this series we build a component around our the thread, so that we can drop it on a form and (Contribute)? it to the TVisualSynapse library.

So, if you're interested in how writing a listening thread with threaded clients, go to the proxy. if you want to learn how to create a component wrapper around a thread, stay here.

In Lesson4 we created a thread. and implemented the procedure. We didn't see how to call the thread. Lets do that first.

```
TfrmMain.button1Click(Sender: TObject);
begin
 //let's create the thread:
 MyTCP := TMyTCPClient.Create (True); //create suspended:
 with MyTCP do
 begin
 //we make the critical section, the thread creates the socket.
 CSData := TCriticalSecion.Create;
 FOnDataReceived := frmMain.OnSomeDatareceived;
 Resume;
 end;
end;
```

notice how we create the critical section here. Creating it in the the execute handler of the thread is not safe, since we may access properties whose handlers need the critical section.
Also, we implement the callback routine. you can implement one yourself by hand, but once we got the component a double-click will do.

LessonProxy

Creating a listening thread and a client handler.

You can download source and binaries from here:
http://synapse.wiki.dubaron.com/files/simpleproxy.zip

Warning: the declaration of the client handler is a bit big piece of code. For a shorter example of a server and a client thread, see CreatingATunnel? or (())

We have the following interface declarations:

```
unit untServer;
interface
uses Windows, Classes, SysUtils, Synautil, blcksock, HTTPSend, untVertaal,
syncobjs;
type
 TServer = class(TThread)
 FSock:TTCPBlockSocket;
 procedure Execute; override;
 end;
 TProxy = class (TThread)
 FSock:TTCPBlockSocket;
 procedure Execute; override;
 end;
```

Now, we set up u listener socket TServer. As you can see, the listener socket will spawn new class of TProxy for each client connection that comes in.

```
procedure TServer.Execute;
var P:TProxy;
begin
 FSock := TTCPBlockSocket.Create;
 FSock.Bind ('0.0.0.0', '8080');
 FSock.Listen;
 if FSock.LastError=0 then
 begin
 while not Terminated do
 begin
 if FSock.CanRead (200) then
 //some client is connecting:
 begin
 P:=TProxy.Create(True);
 P.FSock := TTCPBlockSocket.Create;
 P.FSock.Socket := FSock.Accept;
 //Launch proxy
 P.Resume;
 inc (FNumConnections);
 end
 else
 //;sleep(20);
 end;
 end;
 FSock.CloseSocket;
 FSock.Free;
end;
```

As you can see, the listener socket, after succesfully binding a socket, never checks FLastError?.

- Binding to ('0.0.0.0', someport) will bind to all available IP addresses on the local machine.

You can bind a specific adapter if you want by filling in its ip address.

- if binding fails, the IP/port combination is already in use by another listener socket. In the past, there weere some articles on the mailing list about (multiple listeners on same port)? and (binding multiple ip address)?.

Also, you can only bind one port per socket. If you want to listen on more ports, you'll have to create more sockets. You can check the status of multiple sockets by using the (Select)? method, where you can check multiple sockets at once, instead of using Canread. But all this is theoretical stuff, in practice there is no need to.

The listener server creates client sockets. let's implement one:

```
procedure TProxy.Execute;
var i,j:Integer;
 D:String;
 H:TStringList;
 R:THTTPSend;
 URL, Host, METHOD:String;
 S:String;
 ContentLength:Integer;
begin
 //Read socket
 H:=TStringList.Create;
 while not terminated do
 begin
 if FSock.CanRead (20000) then
 begin
 i:= FSock.WaitingData;
 if i>0 then
 begin
 D:=FSock.RecvString (20000);
 while D<>'' do
 begin
 H.Add(D);
 D:=FSock.RecvString(20000);
 end;
 if H.Count > 0 then //there was _some_ data
 begin
 URL:=H[0];
 if ((pos ('GET ', URL)=1) or
 (pos ('HEAD ', URL)=1) or
 (pos ('POST ', URL)=1) or
 (pos ('TRACE ', URL)=1)) and
 (pos (' HTTP/', URL)>6) then
 begin
 METHOD := copy (URL,1,pos(' ', URL)-1);
 URL:=copy (URL, length(METHOD)+2, maxint);
 URL:=copy (URL, 1, pos(' HTTP/', URL)-1);
 if 'http://'=lowercase(copy(URL,1,7)) then
 begin
 URL:=copy (URL,8,maxint);
 if pos ('/', URL)>1 then
 begin
 Host := Copy (URL, 1, pos('/', URL)-1);
 URL := Copy(URL, pos('/', URL), maxint);
 //Now retrieve the document:
 if (Host<>'') and (URL<>'') then
 begin
 //Build request with HTTPSend
 R:=THTTPSend.Create;
 //Copy headers that might be usefull
 //POST var helper:
```

```
 ContentLength := 0;
 for i:=1 to H.Count - 1 do
 begin
 S:=Copy (H[i], 1, pos (':', H[i])-1);
 if {(S='User-Agent') or} (S='Accept') or (S='Accept-Language') or
 (S='Referer') or (S='If-Modified-Since') or
 (S='If-None-Match') or {(S='Host') or }(S='Cookie') or (S='Pragma') //or
 {(S='Content-Type')} then
 R.Headers.Add (H[i])
 else
 begin
 if (S<>'User-Agent') and
 (S<>'Host') and
 (S<>'Proxy-Connection') and
 (S<>'Content-Length') and
 (S<>'Content-Type') then
 begin
 CSVars.Enter;
 FUnknownHeaders.Add (H[i]);
 CSVars.Leave;
 end;
 end;
 if S='Content-Type' then
 R.MimeType := Copy (H[i], pos (': ', H[i])+2, maxint);
 if S='User-Agent' then
 R.UserAgent := Copy (H[i], pos (': ', H[i])+2, maxint);
 if S='Content-Length' then
 ContentLength := StrToIntDef (Copy (H[i], pos (': ', H[i])+2, maxint), 0);
 // if S='Host' then
// R. Host := Copy (H[i], pos (': ', H[i])+2, maxint);
 end;
 if (METHOD = 'POST') and (ContentLength > 0) then
 //There is some data after the header, read:
 begin
 D:='';
 CSVars.Enter;
 FLastPost.Assign (H);
 CSVars.Leave;
 if (FSock.WaitingData>0) or
 (ContentLength > 0) or
 (FSock.CanRead (6000)) then
 begin
 SetLength (D, ContentLength);
 if FSock.WaitingData > ContentLength then
 SetLength (D, FSock.WaitingData);
 i:=FSock.RecvBufferEx (@D[1], Length(D), 20000);
 if i>ContentLength then
 i:=ContentLength;
 SetLength (D, i);
 FLastPostDataIn.Text := D;
 // D:=D+FSock.RecvBufferStr (128000, 20000);
 end;
 if (D<>'') then
 begin
 //Mark end of data:
// D:=D+#13+#10;
 // R.Protocol := '1.1';
 R.Document.Write (D, Length(D));
 end;
 end;
// R.Headers.Add ('Connection: close');
 if R.HTTPMethod (METHOD, 'http://'+Host+URL) then
 begin
 SetLength (D, R.Document.Size);
```

```
R.Document.Read (D[1], Length(D));
CSVars.Enter;
FURLS.Add ('OK - http://'+Host+URL);
CSVars.Leave;
end
else
begin
CSVars.Enter;
FURLS.Add ('FAILED - http://'+Host+URL);
CSVars.Leave;
R.Headers.Clear;
R.Headers.Add ('HTTP/1.0 404 Not Found');
R.Headers.Add ('Content-Type: text/html');
D:= '<HTML><HEAD><TITLE>404 - Not found</TITLE>'+
'</HEAD><BODY><H1>404 - The page you requested was not found.</H1><BR>'+
'-- Free Translate Proxy server by artee 2003 --'+
'</BODY></HTML>';
R.Headers.Add ('Content-Length: 27419'+IntToStr(Length(D)));
R.Headers.Add ('');
end;
//Now do some action on this document:
//D:=D;
inc (FNumBytes, Length (D));
for i:=0 to R.Headers.Count - 1 do
if pos('Content-Type: text/html', R.Headers[i])=1 then
begin
FContentType.Add (R.Headers[i]);
D:=Translate(D);
end;
j:=-1;
//delete empty headers:
for i:=R.Headers.Count - 1 downto 0 do
if (R.Headers[i]='') then
R.Headers.Delete(i);
//maintain all headers but do adjust content length:
for i:=R.Headers.Count - 1 downto 1 do
begin
if pos('Content-Length:', R.Headers[i])=1 then
begin
j:=i;
// break;
end;
if pos ('Connection: ', R.Headers[i])=1 then
R.Headers.Delete(i);
end;
if R.Headers.Count > 0 then
begin
if j<0 then
j:=R.Headers.Add('');
R.Headers[j] := 'Content-Length: '+IntToStr(Length(D));
R.Headers.Add('Proxy-Connection: close');
R.Headers.Add('Connection: close');
R.Headers.Add('');
FSock.SendString (R.Headers.Text+D);
FSock.CloseSocket;
Terminate;
end;
end;
end;
end;
end
else
//malformed url
begin
```

```
 CSVars.Enter;
 FURLS.Add('?'+URL);
 CSVars.Leave;
 end;
 end;
 // FSock.SendString (D);
 H.Clear;
 end
 else
 Terminate;
 end
 else
 Terminate;
 end;
 H.Free;
 FSock.Free;
end;
```

well, this is a bit dirty code, we should have set some stuff in seperate procedures. The reason that the code is that big is because some things are checked, like httpheaders. Also, it logs quite a lot of information.

What happens:

- The proxy receives some HTTP request.
- If it understands it, a THTTPSend is created. The THTTPSend will retreive the document. Now, at this this stage you can do something with the document. Like translating it, or save it as file. But that is not implented.
- Cookies and other headers are passed trough.

Let's put this all together in this wiki: LessonProxyPutTogether

LessonProxyPutTogether

The unit proxy server put together:

You can download source and binaries from here:
http://synapse.wiki.dubaron.com/files/simpleproxy.zip

```
unit untServer;
interface
uses Windows, Classes, SysUtils, Synautil, blcksock, HTTPSend, untVertaal,
syncobjs;
type
 TServer = class(TThread)
 FSock:TTCPBlockSocket;
 procedure Execute; override;
 end;
 TProxy = class (TThread)
 FSock:TTCPBlockSocket;
 procedure Execute; override;
 end;
//some vars for statistics:
var FNumConnections,
 FNumBytes: Integer;
 FUnknownHeaders:TStrings;
 FLastPost:TStrings;
 FLastPostOut:TStrings;
 FLastPostDataIn:TStrings;
 FLastPostDataOut:TStrings;
 FContentType:TStrings;
 FURLS:TStrings;
 CSVars:TCriticalSection;
implementation
procedure TServer.Execute;
var P:TProxy;
begin
 FSock := TTCPBlockSocket.Create;
 FSock.Bind ('0.0.0.0', '8080');
 FSock.Listen;
 if FSock.LastError=0 then
 begin
 while not Terminated do
 begin
 if FSock.CanRead (200) then
 //some client is connecting:
 begin
 P:=TProxy.Create(True);
 P.FSock := TTCPBlockSocket.Create;
 P.FSock.Socket := FSock.Accept;
 //Launch proxy
 P.Resume;
 inc (FNumConnections);
 end
 else
 //;sleep(20);
 end;
 end;
 FSock.CloseSocket;
 FSock.Free;
end;
procedure TProxy.Execute;
var i,j:Integer;
```

```
 D:String;
 H:TStringList;
 R:THTTPSend;
 URL, Host, METHOD:String;
 S:String;
 ContentLength:Integer;
begin
 //Read socket
 H:=TStringList.Create;
 while not terminated do
 begin
 if FSock.CanRead (20000) then
 begin
 i:= FSock.WaitingData;
 if i>0 then
 begin
 D:=FSock.RecvString (20000);
 while D<>'' do
 begin
 H.Add(D);
 D:=FSock.RecvString(20000);
 end;
 if H.Count > 0 then //there was _some_ data
 begin
 URL:=H[0];
 if ((pos ('GET ', URL)=1) or
 (pos ('HEAD ', URL)=1) or
 (pos ('POST ', URL)=1) or
 (pos ('TRACE ', URL)=1)) and
 (pos (' HTTP/', URL)>6) then
 begin
 METHOD := copy (URL,1,pos(' ', URL)-1);
 URL:=copy (URL, length(METHOD)+2, maxint);
 URL:=copy (URL, 1, pos(' HTTP/', URL)-1);
 if 'http://'=lowercase(copy(URL,1,7)) then
 begin
 URL:=copy (URL,8,maxint);
 if pos ('/', URL)>1 then
 begin
 Host := Copy (URL, 1, pos('/', URL)-1);
 URL := Copy(URL, pos('/', URL), maxint);
 //Now retrieve the document:
 if (Host<>'') and (URL<>'') then
 begin
 //Build request with HTTPSend
 R:=THTTPSend.Create;
 //Copy headers that might be usefull
 //POST var helper:
 ContentLength := 0;
 for i:=1 to H.Count - 1 do
 begin
 S:=Copy (H[i], 1, pos (':', H[i])-1);
 if {(S='User-Agent') or} (S='Accept') or (S='Accept-Language') or
 (S='Referer') or (S='If-Modified-Since') or
 (S='If-None-Match') or {(S='Host') or }(S='Cookie') or (S='Pragma') //or
 {(S='Content-Type')} then
 R.Headers.Add (H[i])
 else
 begin
 if (S<>'User-Agent') and
 (S<>'Host') and
 (S<>'Proxy-Connection') and
 (S<>'Content-Length') and
 (S<>'Content-Type') then
```

```
 begin
 CSVars.Enter;
 FUnknownHeaders.Add (H[i]);
 CSVars.Leave;
 end;
 end;
 if S='Content-Type' then
 R.MimeType := Copy (H[i], pos (': ', H[i])+2, maxint);
 if S='User-Agent' then
 R.UserAgent := Copy (H[i], pos (': ', H[i])+2, maxint);
 if S='Content-Length' then
 ContentLength := StrToIntDef (Copy (H[i], pos (': ', H[i])+2, maxint), 0);
 // if S='Host' then
// R. Host := Copy (H[i], pos (': ', H[i])+2, maxint);
 end;
 if (METHOD = 'POST') and (ContentLength > 0) then
 //There is some data after the header, read:
 begin
 D:='';
 CSVars.Enter;
 FLastPost.Assign (H);
 CSVars.Leave;
 if (FSock.WaitingData>0) or
 (ContentLength > 0) or
 (FSock.CanRead (6000)) then
 begin
 SetLength (D, ContentLength);
 if FSock.WaitingData > ContentLength then
 SetLength (D, FSock.WaitingData);
 i:=FSock.RecvBufferEx (@D[1], Length(D), 20000);
 if i>ContentLength then
 i:=ContentLength;
 SetLength (D, i);
 FLastPostDataIn.Text := D;
 // D:=D+FSock.RecvBufferStr (128000, 20000);
 end;
 if (D<>'') then
 begin
 //Mark end of data:
// D:=D+#13+#10;
 // R.Protocol := '1.1';
 R.Document.Write (D, Length(D));
 end;
 end;
// R.Headers.Add ('Connection: close');
 if R.HTTPMethod (METHOD, 'http://'+Host+URL) then
 begin
 SetLength (D, R.Document.Size);
 R.Document.Read (D[1], Length(D));
 CSVars.Enter;
 FURLS.Add ('OK - http://'+Host+URL);
 CSVars.Leave;
 end
 else
 begin
 CSVars.Enter;
 FURLS.Add ('FAILED - http://'+Host+URL);
 CSVars.Leave;
 R.Headers.Clear;
 R.Headers.Add ('HTTP/1.0 404 Not Found');
 R.Headers.Add ('Content-Type: text/html');
 D:= '<HTML><HEAD><TITLE>404 - Not found</TITLE>'+
 '</HEAD><BODY><H1>404 - The page you requested was not found.</H1><BR>'+
 '-- Free Translate Proxy server by artee 2003 --'+
```

```
'</BODY></HTML>';
R.Headers.Add ('Content-Length: 25493'+IntToStr(Length(D)));
R.Headers.Add ('');
end;
//Now do some action on this document:
//D:=D;
inc (FNumBytes, Length (D));
for i:=0 to R.Headers.Count - 1 do
if pos('Content-Type: text/html', R.Headers[i])=1 then
begin
FContentType.Add (R.Headers[i]);
D:=Translate(D);
end;
j:=-1;
//delete empty headers:
for i:=R.Headers.Count - 1 downto 0 do
if (R.Headers[i]='') then
R.Headers.Delete(i);
//maintain all headers but do adjust content length:
for i:=R.Headers.Count - 1 downto 1 do
begin
if pos('Content-Length:', R.Headers[i])=1 then
begin
j:=i;
// break;
end;
if pos ('Connection: ', R.Headers[i])=1 then
R.Headers.Delete(i);
end;
if R.Headers.Count > 0 then
begin
if j<0 then
j:=R.Headers.Add('');
R.Headers[j] := 'Content-Length: '+IntToStr(Length(D));
R.Headers.Add('Proxy-Connection: close');
R.Headers.Add('Connection: close');
R.Headers.Add('');
FSock.SendString (R.Headers.Text+D);
FSock.CloseSocket;
Terminate;
end;
end;
end;
end
else
//malformed url
begin
CSVars.Enter;
FURLS.Add('?'+URL);
CSVars.Leave;
end;
end;
// FSock.SendString (D);
H.Clear;
end
else
Terminate;
end
else
Terminate;
end;
H.Free;
FSock.Free;
```

```
end;
initialization
 FUnknownHeaders := TStringList.Create;
 FLastPost := TStringList.Create;
 FURLS:=TStringList.Create;
 FLastPostOut:=TStringList.Create;
 FLastPostDataIn:=TStringList.Create;
 FLastPostDataOut:=TStringList.Create;
 FContentType := TStringList.Create;
 CSVars := TCriticalSection.Create;
finalization
 FUnknownHeaders.Free;
 FLastPost.Free;
 FURLS.Free;
 CSVars.Free;
end.
```

Appendix G – Visual Synapse

TVisualSynapse

TVisualSynapse is the base class for most visual synapse components.

All synapse components based on TVisualSynapse will inherit these properties. In some cases some properties may be hided or obsolete, depending on specific component behaviour.

Properties

- SocksProxyInfo? - links to a TSocksProxyInfo? component
- MaxThreads - number of maximum allowed threads and thus number of maximum concurrent connection
- SendBandWidth?, RecvBandWidth? - if non-zero, limits the bandwith for outgoing resp. incoming traffic. Bytes/second
- LastJob? - specifies the identifier of last job.

Methods

TVisualSynapse does not define any public methods.

Events

- OnData?:TOnVisualData
    - occurs when incoming data is available. Data is passed as string
- property OnDataStrings?:TOnDataStrings
    - The same as OnData?, except that data is passed as a TStrings instead of a String
- OnError?:TOnError read FOnError? write FOnError?;
    - If an error occurs, the error id and an error message is passed.
- OnProgress?:TOnProgress
    - Callback function allowing to monitor connection progress. OnProgess? is called each time a synapse socket calls a TOnHookScoket?. OnProgress? will be called each time one of the following events is triggered:
        - OnResolvingBegin?:TOnSockStatus
        - OnResolvingEnd?:TOnSockStatus
        - OnSocketCreate?:TOnSockStatus
        - OnSocketClose?:TOnSockStatus
        - OnBind?:TOnSockStatus
        - OnConnect?:TOnSockStatus
        - OnCanRead?:TOnSockStatus
        - OnCanWrite?:TOnSockStatus
        - OnListen?:TOnSockStatus
        - OnAccept?:TOnSockStatus
        - OnReadCount?:TOnSockInteger
            - Count the number of bytes received
        - OnWriteCount?:TOnSockInteger
            - Count the number of bytes sent
        - OnWait?:TOnSockStatus
        - OnSockError?:TOnSockStatus

Hidden properties

Most synapse classes are clients, and do not need the OnListen? and OnAccept? events. Therefore an extra class, TVisualClient, is declared. This is the same class as TVisualSynapse, but hides obsolete properties.

Underlying behaviour:

TVisualSynapse arranges the job queueing. Each time some request is made by the component, like a connection request, sending data, retrieving a html, sending an e-mail etc, a object of class TJob is created. The TJob is queued in a list of jobs.
If all threads are busy, and the number of threads is less then MaxThreads, a new thread will be created.
Threads will poll the job queue. If a thread is ready to perform a new command, it will delete the job from the ueue and perform the task.
On creation of a job, each job (belonging to a specific component) is given a unique identifier. This identifier is always passed on the callback events. Methods that will create a new job will return a job identifier. You can also retrieve the job identifier of the last created job by reading the LastJob? property. All jobs, valid or invalid, will return a job identifier. If a job was invalid or unsuccesfull, a OnError? callback will occur.
Derivants of TVisualSynapse wil mostly also derive a customized TJob class. The TJob class will contain all information that is needed to perform a specific task.

Appendix H – Clamav

Warning – this section is outdated. To use clamav it is recommended to use a clamav server and the appropiate synapse clamav (TClamSend) library.

However the virus scanner itself is a nice example of i real quick scanner that maps virus signatures in a binary tree. Works with obsolete clamav AV files only. Real-life tested on email attachments.

ClamavAntivirusDelphi

Abstract

How to implement clamav antivirus support with delphi

Description

This document will provide you all necessary information and delphi source code to implement clamav antivirus support with delphi or kylix.
An example is included how to fetch and decode a mime message from a pop3 or imap server using synapse, and then scan it.

Before you begin

You will need to fetch the following clamav definitions. Watch the timestamp to see if you need to update.
http://database.clamav.net/database/timestamp
http://database.clamav.net/database/viruses.db
http://database.clamav.net/database/viruses.db2

Loading the virus definitions in the antivirus unit

do this for every virus definition you have:

```
TAntiVirus.Load (FileName);
```

The antivirus unit

Origin of this unit: http://sourceforge.net/projects/eemailer
License: Modified Artistic License

```
unit unitAntiViri;
// Free for any legal use see Modified Artistic License
// By Rene Tegel 2004
// Fast Clamav antivirus scan implementation
// Tested with delphi
interface
uses Classes, SysUtils, syncobjs, windows;
type TViriDef = class (TObject)
 name:String;
 Definition:String;
 SubDefs:TStrings;
 end;
type TAntiViri = class (TObject)
 protected
 class procedure Clear;
 public
// constructor Create;
// destructor Destroy;
 class procedure Load (FileName:String);
 class function IsVirus (Value:String):String;
 end;
```

```
implementation
//global
type TLookUp = array [0..$FFFF] of TList;
var
 FVira:TList;
 FCS: TCriticalSection;
 Lookup: TLookup;
 //TList(Lookup[..]) contains TViriDef entries
 FAC: Integer = 0;
 FWait: Boolean = False;
{ TAntiViri }
class procedure TAntiViri.Clear;
var i:Integer;
begin
 for i:=0 to FVira.Count - 1 do
 TObject(FVira[i]).Free;
 FVira.Clear;
end;
class function TAntiViri.IsVirus(Value: String): String;
//This is not a perfect implementation.
//We check if additional occurences occur, if any, but do not test on location
//see *.db2 files and the format of those definitions.
//This is relative easy to implement
//but outside the scope of this demo.
//main part is the lookup table to demonstrate how to do
//a high performance scan.
var i, j, n, //counters
 l, //length of data
 li:Integer; //lookup index
 Found: Boolean; //helper var for virus definitions with multiple entries
begin
 while FWait do
 sleep (0);
 FCS.Enter;
 inc (FAC);
 FCS.Leave;
 try
 {using lookup table}
 i := 1;
 l := length (Value);
 Result := '';
 while i < l do
 begin
 li := byte (Value[i]) shl 8 or byte (Value[i+1]);
 if Assigned (lookup [li]) then
 begin
 //check description
 for j := 0 to Lookup[li].Count - 1 do
 with TViriDef (Lookup[li].Items[j]) do
 begin
// if copy (Value, i, length (Definition)) = Definition then
 //avoid string copy, just compare memory :)
 if (i+length(Definition)<=l) and
 (CompareMem (@Value[i], Pointer(Definition), length (Definition)) ) then
 begin
 //see if there are additional references declared
 if Assigned (SubDefs) and (SubDefs.Count > 0) then
 begin
 //check them
 Found := true;
 for n := 0 to SubDefs.Count - 1 do
 begin
 if pos (SubDefs[n], Value)=0 then
 begin
```

```
 Found := False;
 break;
 end;
 end;
 if Found then
 begin
 Result := Name;
 break;
 end;
 end
 else
 //./ this was a hit :)
 begin
 Result := Name;
 break;
 end;
 end;
 end;
 end;
 if Result <> '' then
 break;
 inc (i);
 end;
 except end;
 FCS.Enter;
 dec (FAC);
 FCS.Leave;
end;
class procedure TAntiViri.Load;
var i,j,li,n:Integer;
 v:TViriDef;
 h,s,w:String;
 FVirusses:TStrings;
 Defs: TStrings;
begin
 //load virus list from file
 FWait := True;
 for i:=0 to 200 do
 sleep (0);
 //now, wait until all access is done
 while (FAC > 0) do
 begin
 sleep (0);
 end;
 try
 FVirusses := TStringList.Create;
 FCS.Enter;
 try
 FVirusses.LoadFromFile (FileName);
 Defs := TStringList.Create;
 //only clear old if loading fails
// Clear;
 for i:=0 to FVirusses.Count - 1 do
 begin
 v := TViriDef.Create;
 v.name := FVirusses.Names [i];
 h := FVirusses[i];
 h := copy (h, pos('=', h)+1, maxint);
 h := stringreplace (h, '*', '_', [rfReplaceAll]);
 h := stringreplace (h, '?', '_', [rfReplaceAll]);
 w := stringreplace (h, '__', '_', [rfReplaceAll]);
 //filter out doubles:
 while length (w) < length (h) do
 begin
```

```
 h := w;
 w := stringreplace (h, '__', '_', [rfReplaceAll]);
 end;
 Defs.Text := stringreplace (w, '_', #13#10, [rfReplaceAll]);
 if Defs.Count = 0 then
 continue;
 s := '';
 j := 1;
 while j < length(Defs[0]) do
 begin
 s := s + char(StrToIntDef ('$'+copy (h,j,2), 0));
 inc (j, 2);
 end;
 v.Definition := s;
 FVira.Add (v);
 if Defs.Count > 1 then
 begin
 v.SubDefs := TStringList.Create;
 for n := 1 to Defs.Count - 1 do
 begin
 j := 1;
 s := '';
 while j < length(Defs[n]) do
 begin
 s := s + char(StrToIntDef ('$'+copy (Defs[n],j,2), 0));
 inc (j, 2);
 end;
 v.SubDefs.Add (s);
 end;
 end;
 //now add an entry to Lookup
 if length(v.Definition)>2 then
 begin
 li := byte(v.Definition[1]) shl 8 or byte(v.Definition[2]);
 if not Assigned (Lookup[li]) then
 Lookup[li] := TList.Create;
 Lookup[li].Add (v);
 end;
 end;
 finally
 FCS.Leave;
 FVirusses.Free;
 end;
 except end;
 FWait := False;
end;
var i:Integer;
initialization
 FVira := TList.Create;
 FCS := TCriticalSection.Create;
 for i:=0 to high (Lookup) do
 Lookup[i] := nil;
finalization
 //clear FVira list
 TAntiViri.Clear;
 FVira.Free;
 FCS.Free;
end.
```

How to check a mime message

```
function CheckVirus (MP: TMimePart): String;
var i:Integer;
 n,v:String;
begin
 // loop all parts
 // if virus found, result := virusname
 // and break
 Result := '';
 try //no particulair reason, just safety
 for i := 0 to MP.GetSubPartCount -1 do
 begin
 Result := checkVirus (MP.GetSubPart (i));
 if Result <> '' then
 break;
 end;
 MP.DecodePart;
 setlength (n, MP.DecodedLines.Size);
 MP.DecodedLines.Read (n[1], length(n));
 v := TAntiViri.IsVirus (n);
 if (v<>'') then
 begin
 if Result <> '' then
 Result := Result + ';' + v
 else
 Result := v;
 end;
 except end;
end;
```

How to fetch and check a mime message

```
 IMAP.FetchMess (uid, Mime.Lines);
 Mime.DecodeMessage;
 if CheckVirus (Mime.MessagePart)<>'' then
 // virus found
```