

网络攻防基础 实验 1A 网络嗅探器的设计与实现

*** 2023180186*****

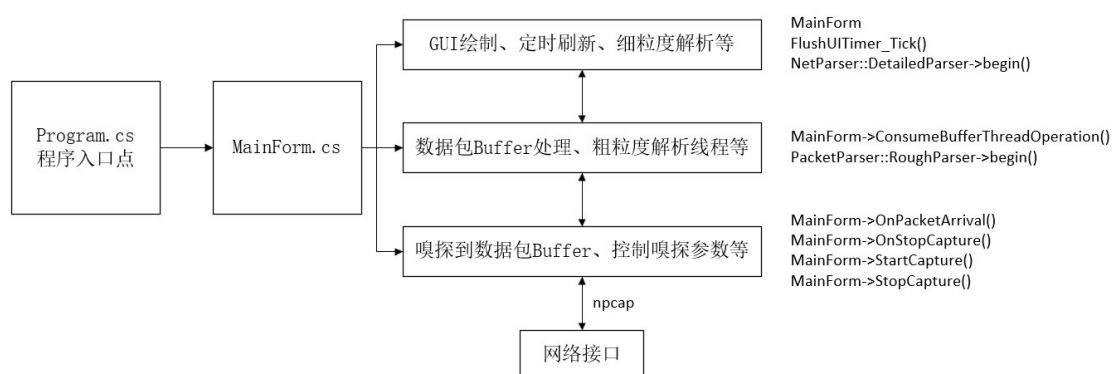
软件简介

实验要求设计一个网络嗅探器，要求实现网络嗅探功能，并尽量完善网络分析功能。本网络嗅探器取名为 NetHarbor，开发工具使用 Visual Studio 2022，整体基于 C#语言和 .NET Framework 开发，图形界面基于 WinForm 技术，捕获网络流量使用了 Npcap（开启 Winpcap 兼容模式）提供的 API。

本软件提供了网络接口获取、捕获过滤器和网络流量捕获与捕获分组数统计功能，支持对 Ethernet 帧、ARP 协议、IPv4/IPv6 协议、ICMP/ICMPv6 协议、TCP 协议、UDP 协议以及 TLS 协议、HTTP 协议和 DNS 协议数据包的简单解析。

软件总体设计与代码结构

软件总体架构原理简化如下。

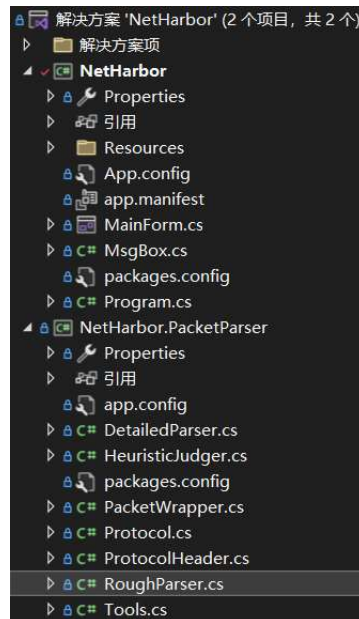


本软件项目源码结构目录如下图所示。

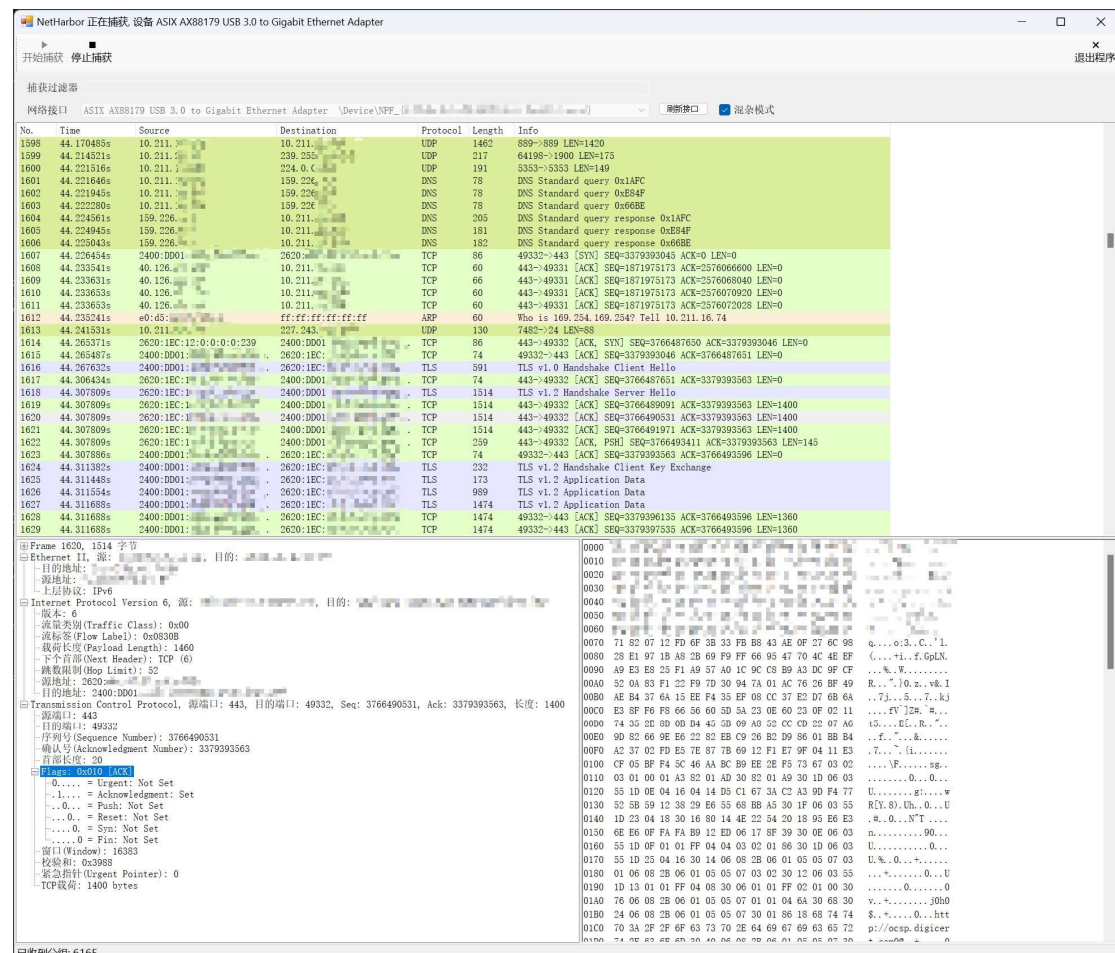
NetHarbor 为项目主目录，Program.cs 为项目入口点，MainForm.cs 有 GUI 处理相关、包嗅探与控制相关代码以及包粗粒度处理解析线程相关代码，MsgBox.cs 为封装后的 MessageBox 控件相关方法。

NetHarbor.PacketParser 为解析相关模块代码目录，其中 RoughParser.cs 和 DetailedParser.cs 分别为粗粒度包解析器和细粒度包解析器，分别用于处理显示信息、初步解析数据包协议和包详细信息分析与展示。PacketWrapper.cs 是数据包包装类，用于封装存储与数据包相关的内容。Protocol.cs 和 ProtocolHeader.cs 分别为协议与协议详细内容枚举类和协议首部结构体类，其中存放了与协议定义有关的内容，方便解析器使用。HeuristicJudger.cs 存放了启发式解析规则判断的相关内容，具体内容在下文“关键内容梳理”中详细描述。

Tools.cs 包含与网络数据包处理相关的工具类，如大端序转小端序、二进制数据转 IP 字符串&MAC 字符串、从指针中拷贝定长数据等功能。



软件主界面如图所示。



软件关键内容梳理

1、在 C#中为提高效率，采用 unsafe 关键字与 fixed 关键字将部分代码设置为不安全代码并固定指针，可以直接操作相关指针，同时采用结构体的处理方式，进一步提升了数据包解析和处理的效率。

2、通过设置包 buffer 机制，并将包处理解析线程和 GUI 处理线程分离开，再配合粗粒度解析包时避免耗时操作，能够避免在较高速通信环境下大量包涌入时程序失去响应或包在 buffer 中堆积无法处理。

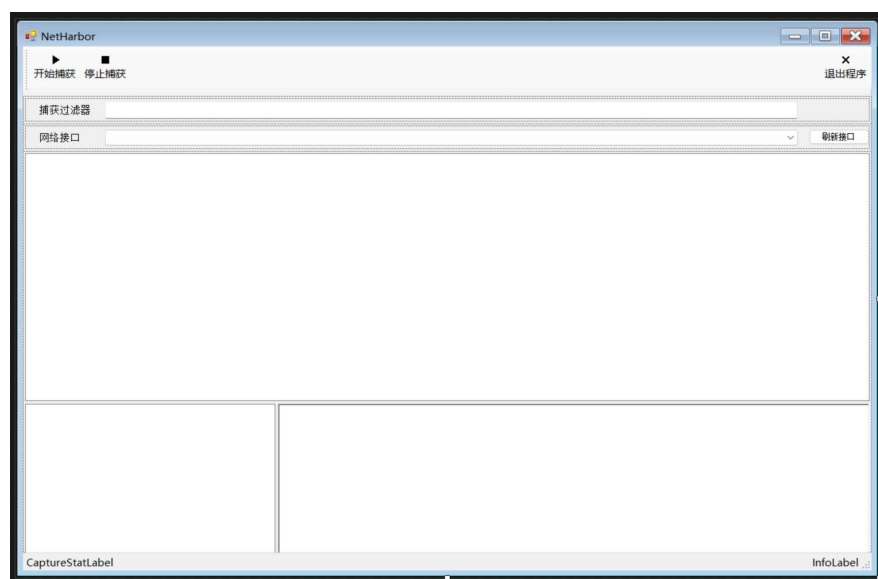
3、在 GUI 界面主要的包列表浏览 ListView 控件绘制中，不再采用一次性全部加载的传统模式，而是采用“懒加载”方式 (VirtualMode)，在用户浏览到对应区段的包时才构造、加载对应的内容，避免包的大量涌入让 ListView 控件无法处理，进而导致 GUI 界面卡死无法响应的情况出现。同时，GUI 界面捕获分组统计信息的显示也采取了定时刷新的机制，在实时性和处理效率之间取得了一定平衡。

4、进行数据包内容解析时，在包 buffer 处理线程中进行粗粒度解析，取出最低限度的信息，随后在用户查看数据包详细信息时，再对其余需要解析的部分进行读取。两者配合，总体来说节省了处理时间，提高了软件效率。

5、针对如何判断应用层协议的问题，通过翻阅源码和参考资料后，借鉴“启发式解析”算法，定义特定规则，实现了对部分应用层协议的简单解析，具体在“HeuristicJuder.cs”中有所定义。具体实现请见“协议解析具体设计”部分，具体说明请见“实验遇到的问题和实验总结-思考题”部分。

界面设计与刷新具体设计

程序界面设计如图所示。



最上方是一个菜单栏，中间使用 Button 控件对功能进行控制，接着是两个 Panel 控件，里面通过 TextBox、ComboBox、CheckBox 和 Button 控件实现了相关的选择和输入功能。接着是一个 ListView 控件，主要用于显示包的信息，下方被分割成两个区域，左侧放置 TreeView 控件，用于包详细内容的展示，右侧放置 RichTextBox 控件，用于显示包 16 进制原始信息。最底部是一个 StatusStrip 控件，里面通过 Label 控件显示软件当前的运行状态和统计信息。

在程序代码的控制上，主要是对控件属性的控制和事件的绑定，因为主要代码实现均在“MainForm.cs”中，逻辑较为简单，注释较为完善，在此仅详细描述 ListView 的“懒加载”如何实现，其余代码逻辑不再赘述。。

在 MainForm 的构造函数中，可以看到绑定了 ListView 懒加载的模式和加载事件。

```
1 个引用
public MainForm()
{
    InitializeComponent();
    StopCaptureButton.Enabled = false;
    DoubleBuffered = true;
    // ListView懒加载
    PacketListView.VirtualMode = true;
    PacketListView.RetrieveVirtualItem += PacketListView_RetrieveVirtualItem;
    // UI刷新计时器
    FlushUITimer.Enabled = true;
    FlushUITimer.Interval = 250;
}
```

主界面加载完成事件中，可以看到 ListView 加载列名的相关代码。

```
//主界面加载完毕事件
1 个引用
private void MainForm_Load(object sender, EventArgs e)
{
    // 界面加载后载入接口列表
    RefreshInterfaceButton.PerformClick();
    // 载入ListView列名
    PacketListView.View = View.Details;
    PacketListView.Columns.Add("No.", 70);
    PacketListView.Columns.Add("Time", 140);
    PacketListView.Columns.Add("Source", 250);
    PacketListView.Columns.Add("Destination", 250);
    PacketListView.Columns.Add("Protocol", 90);
    PacketListView.Columns.Add("Length", 80);
    PacketListView.Columns.Add("Info", 550);
    // 渲染Splitter
    splitContainer2.SplitterDistance = this.Width / 2;
    InfoLabel.Text = "";
}
```

最后，可以看到 ListView 加载虚拟模式项的相关代码。

```
1 个引用
private void PacketListView_RetrieveVirtualItem(object sender, RetrieveVirtualItemEventArgs e)
{
    readList = packetList;
    if (e.ItemIndex >= 0 && e.ItemIndex < readList.Count)
    {
        PacketWrapper pkt = readList[e.ItemIndex];
        ListViewItem listViewItem = new ListViewItem();
        listViewItem.Text = pkt.Count.ToString();
        listViewItem.SubItems.Add(pkt.RelativeTimeval.ToString());
        listViewItem.SubItems.Add(pkt.TopSource.ToString());
        listViewItem.SubItems.Add(pkt.TopDestination.ToString());
        listViewItem.SubItems.Add(pkt.TopProtocol.ToString());
        listViewItem.SubItems.Add(pkt.Length.ToString());
        listViewItem.SubItems.Add(pkt.TopInfo.ToString());
        listViewItem.BackColor = GetProtocolColor(pkt.TopProtocol);
        // 提供虚拟项的数据
        e.Item = listViewItem;
    }
}
```

流量捕获具体设计

流量捕获主要借助了 npcap 和 sharppcap 提供的相关捕获 API，下面按照实现逻辑介绍具体的代码实现。

- 1、获取接口列表并显示在 ComboBox 中。

```
//刷新接口按钮点击事件
1 个引用
private void RefreshInterfaceButton_Click(object sender, EventArgs e)
{
    InterfaceComboBox.Items.Clear();
    var devices = LibPcapLiveDeviceList.Instance;
    if (devices.Count < 1)
    {
        MessageBox.ShowError("没有发现可用的网络接口!");
        return;
    }
    foreach (var dev in devices)
        InterfaceComboBox.Items.Add(String.Format("{0} {1}", dev.Description, dev.Name));
}
```

- 2、用户操作点击开始捕获后，调整相关变量、捕获参数（混杂模式和捕获过滤器等），并绑定数据包到达事件。

```
1 个引用
private void StartCapture(int interfaceIndex, bool promiscuousMode, string filter)
{
    device = LibPcapLiveDeviceList.Instance[interfaceIndex];
    // device = CaptureDeviceList.Instance[interfaceIndex];
    packetCount = 0; // 初始化包序号
    firstPacketTimeval = null; // 初始化第一个包时间
    PacketListView.VirtualListSize = 0; // 清除VirtualMode的计数否则会出问题
    PacketListView.Items.Clear(); // 确保清除了计数再清空项目渲染
    packetList = new List<PacketWrapper>(); // 然后再清除packetList
    readList = packetList;

    // 开启设备，绑定事件
    arrivalEventHandler = new PacketArrivalEventHandler(OnPacketArrive);
    device.OnPacketArrival += arrivalEventHandler;
    captureStoppedEventHandler = new CaptureStoppedEventHandler(OnCaptureStop);
    device.OnCaptureStopped += captureStoppedEventHandler;

    if (promiscuousMode)
    {
        device.Open(DeviceModes.Promiscuous);
    }
    else
    {
        device.Open(DeviceModes.None);
    }

    try
    {
        if (filter != null && filter.Length > 0)
        {
            device.Filter = filter;
        }
    }
    catch (Exception)
    {
        MessageBox.ShowError(String.Format("捕获过滤器: {0} 不合法!", filter));
        // 卸载事件
        device.OnPacketArrival -= arrivalEventHandler;
        device.OnCaptureStopped -= captureStoppedEventHandler;
        return;
    }
}
```

- 3、开启包 Buffer 处理线程，开启设备，并同步绘制 GUI。


```

// 统计数据刷新
captureStatistics = device.Statistics;

// 开启包buffer处理线程
stopConsumeBufferThread = false;
consumeBufferThread = new System.Threading.Thread(ConsumeBufferThreadOperation);
consumeBufferThread.Start();

// 开启捕获
device.StartCapture();

//GUI绘制
StartCaptureButton.Enabled = false;
StopCaptureButton.Enabled = true;
InterfaceComboBox.Enabled = false;
CaptureFilterTextbox.Enabled = false;
this.Text = String.Format("NetHarbor 正在捕获, 设备 {0}", device.Description);
}

```

继续实现包到达事件。主要思路为，到达包后，直接送入包 Buffer 中进行缓存，等待包处理线程对 buffer 进行处理。采用 buffer 可以对包进行缓存，防止乱序，提高处理的简便性。同时，按照时间间隔读取统计数据，以便 UI 刷新计时器对数据同步更新绘制到 GUI 上。在这个过程中，需要使用锁机制，防止多线程的并发竞争问题。

```

void OnPacketArrive(object sender, PacketCapture e)
{
    // 将包存入buffer, buffer机制可以防止包乱序, lock住防止多线程搞事情
    lock(bufferLock)
    {
        packetBuffer.Enqueue(e.GetPacket());
    }

    // 按时间间隔读取统计数据
    var NowTime = DateTime.Now;
    if ((NowTime - LastStatisticsTime) > LastStatisticsInterval)
    {
        captureStatistics = e.Device.Statistics;
        LastStatisticsTime = NowTime;
    }
}

```

下面介绍包 Buffer 处理线程的具体实现。

- 1、采用 while 循环执行本线程的操作，若判断到停止信号，则不再执行循环，停止处理线程。
- 2、新建一个 Buffer 队列，并更换包捕获事件中正在工作的 Buffer 队列到这个新建的队列上，方便后续对 buffer 内的数据包进行处理。注意过程中需要使用锁机制避免多线程并发竞争。

```

while (!stopConsumeBufferThread)
{
    // List<RawCapture> processQueue = null;
    Queue<RawCapture> processQueue = null;
    // 检查Buffer中是否有包
    lock (bufferLock)
    {
        if (packetBuffer.Count != 0)
        {
            processQueue = packetBuffer;
            //packetBuffer = new List<RawCapture>();
            packetBuffer = new Queue<RawCapture>();
        }
    }
}

```

- 3、对刚刚拿到的 Buffer 利用粗粒度解析器进行处理，并计算与第一个包的相对时间差。包处理后将其加入到处理后列表 packetList。若 buffer 中无内容，则跳过本次处理。

```

if(processQueue == null)
{
    System.Threading.Thread.Sleep(0);
}
else
{
    Console.WriteLine(String.Format("Consuming... Buffer Len: {0}", processQueue.Count));

    // 已经处在上一个if的else中, 一定有一个包, 无需判空
    while (processQueue.Count > 0)
    {
        raw = processQueue.Dequeue();
        // 时间所限, 无法处理非Ethernet
        if (raw.LinkLayerType != LinkLayers.Ethernet)
            continue;
        packetCount++;
        var packetWrapper = new PacketWrapper(packetCount, raw, firstPacketTimeval);
        RoughParser.begin(packetWrapper);
        packetList.Add(packetWrapper);
    }

    if (firstPacketTimeval == null && packetList.Count >= 1)
    {
        firstPacketTimeval = packetList[0].Timeval;
    }

    Console.WriteLine(packetList.Count);
}

```

协议解析具体设计

协议解析中，主要分为粗粒度解析和细粒度解析两种，分别在“NetHarbor.PacketParser”项目中“RoughParser.cs”和“DetailedParser.cs”中有所体现。两者都提供了 begin() 方法对传入的数据包开始不同层次粒度的解析。同时，还编写了简要的启发式解析规则、枚举定义、协议内容定义、协议首部结构体并利用了 C# 中提供的 unsafe 关键字和 fixed 关键字用于直接操作指针，以便更加高效和快捷的对数据包进行操作。

首先介绍粗粒度解析器的相关设计与实现。

1、begin 方法，用于固定指针，并准备好数据传入到链路层解析方法 LinkLayer 中。注意方法中用到的 unsafe 关键字和 fixed 关键字。

```

public unsafe static void begin(PacketWrapper pkt)
{
    fixed (byte* ptr = pkt.Data)
    {
        LinkLayer(pkt, ptr);
    }
}

```

2、链路层解析方法 LinkLayer，用于解析链路层相关的信息，并判断上层的协议。注意定义的 EthernetHeader 以太网首部结构体在其中的应用。

```

public unsafe static void LinkLayer(PacketWrapper pkt, byte* ptr)
{
    EthernetHeader eth = *(EthernetHeader*) ptr;
    switch (Tools.NetworkToHost(eth.type))
    {
        case 0x0800:
            pkt.NetworkLayerProtocol = NetworkLayerProtocol.IP;
            break;
        case 0x0806:
            pkt.NetworkLayerProtocol = NetworkLayerProtocol.ARP;
            break;
        case 0x86DD:
            pkt.NetworkLayerProtocol = NetworkLayerProtocol.IPv6;
            break;
        default:
            pkt.NetworkLayerProtocol = NetworkLayerProtocol.Other;
            break;
    }
    pkt.TopProtocol = pkt.NetworkLayerProtocol.ToString();
    pkt.LinkLayerSource = Tools.GetMacHexFromByteArray(eth.sourceAddress);
    pkt.LinkLayerDestination = Tools.GetMacHexFromByteArray(eth.destinationAddress);
    pkt.TopDestination = pkt.LinkLayerDestination;
    pkt.TopSource = pkt.LinkLayerSource;
    pkt.TopInfo = "Ethernet";
    NetworkLayer(pkt, ptr+sizeof(EthernetHeader));
}

```

3、网络层解析方法 NetworkLayer，用于解析网络层相关信息，放入 PacketWrapper 中。由于代码较长，仅以 IP 协议为例。其余请参见源码。

```

public unsafe static void NetworkLayer(PacketWrapper pkt, byte* ptr)
{
    switch (pkt.NetworkLayerProtocol)
    {
        case NetworkLayerProtocol.IP:
            IPHeader ip = *(IPHeader*)ptr;
            switch (ip.protocol)
            {
                // 此处理并非属于传输层协议
                // 只是方便管理，ICMP、IGMP等本身传递一定信息，且后没有其他传输层payload
                // 具体见Protocol.cs文件中注释
                case 1:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.ICMP;
                    break;
                case 2:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.IGMP;
                    break;
                case 6:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.TCP;
                    break;
                case 17:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.UDP;
                    break;
                case 41:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.IPv6OverIPv4;
                    break;
                default:
                    pkt.TransmissionLayerProtocol = TransmissionLayerProtocol.Other;
                    break;
            }
            pkt.NetworkLayerSource = Tools.GetIPFromBytes(ip.sourceAddress);
            pkt.NetworkLayerDestination = Tools.GetIPFromBytes(ip.destinationAddress);
            pkt.TopSource = pkt.NetworkLayerSource;
            pkt.TopDestination = pkt.NetworkLayerDestination;
            pkt.TopProtocol = pkt.TransmissionLayerProtocol.ToString();
            pkt.TopInfo = "IP";
            pkt.NetworkLayerHeaderLength = (4 * (ip.versionAndHeadLength & 0x0F));
            pkt.NetworkLayerPayloadLength = Tools.NetworkToHost(ip.totalLength);
            // 取出HeadLength，并乘以4跳过IP头。
            TrasmissionLayer(pkt, ptr + pkt.NetworkLayerHeaderLength);
            break;
        case NetworkLayerProtocol.ARP:
            ARPHeader arp = *(ARPHeader*)ptr;

```


4、传输层协议解析 TransmissionLayer 方法，该方法同样是解析包中对应协议的内容，并存入 PacketWrapper 中。可以注意到该方法中使用了位运算对数据包进行解析，操作效率较高。同时，通过代码可以看到，TCP 解析中使用 IP 总长度、IP 首部长度和 TCP 首部长度的可以计算出 TCP 载荷长度。由于代码较长，仅展示 TCP 协议部分处理代码。

```
switch(pkt.TransmissionLayerProtocol)
{
    case TransmissionLayerProtocol.TCP:
        TCPHeader tcp = *(TCPHeader*)ptr;
        int tcpHeaderLength = (tcp.headerLength >> 4) * 4;
        pkt.TransmissionLayerPayloadLength = pkt.NetworkLayerPayloadLength - pkt.NetworkLayerHeaderLength - tcpHeaderLength;
        // Console.WriteLine("{0} {1} {2}", pkt.NetworkLayerPayloadLength, pkt.NetworkLayerHeaderLength, tcpHeaderLength);
        pkt.TransmissionSourcePort = Tools.NetworkToHost(tcp.sourcePort);
        pkt.TransmissionDestinationPort = Tools.NetworkToHost(tcp.destinationPort);
        string flag_str = "";
        if((tcp.flags & 0x20) >> 5 == 1)
        {
            flag_str += "URG, ";
        }
        if((tcp.flags & 0x10) >> 4 == 1)
        {
            flag_str += "ACK, ";
        }
        if ((tcp.flags & 0x08) >> 3 == 1)
        {
            flag_str += "PSH, ";
        }
        if ((tcp.flags & 0x04) >> 2 == 1)
        {
            flag_str += "RST, ";
        }
        if ((tcp.flags & 0x02) >> 1 == 1)
        {
            flag_str += "SYN, ";
        }
        if ((tcp.flags & 0x01) == 1)
        {
            flag_str += "FIN, ";
        }
        if (flag_str.Length > 0)
        {
            flag_str = flag_str.Substring(0, flag_str.Length - 2);
        }

        uint seqNumber = Tools.NetworkToHost(tcp.sequence);
        uint ackNumber = Tools.NetworkToHost(tcp.acknowledgement);
        pkt.TopInfo = String.Format("{0}->{1} [{2}] SEQ={3} ACK={4} LEN={5}",
            pkt.TransmissionSourcePort, pkt.TransmissionDestinationPort, flag_str,
            seqNumber, ackNumber, pkt.TransmissionLayerPayloadLength);
        ApplicationLayer(pkt, ptr + tcpHeaderLength);
        break;
    case TransmissionLayerProtocol.UDP:
        UDPHeader udp = *(UDPHeader*)ptr;
```

5、应用层协议解析 ApplicationLayer 方法，该方法同样是解析包中对应协议的内容，并存入 PacketWrapper 中。该方法亮点是采用了简要启发式规则对 TLS 和 HTTP 流量进行解析。DNS 协议由于时间和任务量原因，仅通过端口号进行判断解析。

```

public unsafe static void ApplicationLayer(PacketWrapper pkt, byte* ptr)
{
    switch (pkt.TransmissionLayerProtocol)
    {
        case TransmissionLayerProtocol.TCP:
            // 解析TCP payload
            // 启发式解析判断应用层协议类型
            if (HeuristicJuderger.IsTlsProtocol(ptr))
            {
                // 是TLS
                pkt.ApplicationLayerProtocol = ApplicationLayerProtocol.TLS;
                pkt.TopProtocol = pkt.ApplicationLayerProtocol.ToString();

                byte contentType = *ptr;
                ptr++;
                ushort* pointer = (ushort*)ptr;
                ushort version = Tools.NetworkToHost(*pointer);

                ptr += sizeof(ushort);
                pointer++;
                ushort length = Tools.NetworkToHost(*pointer);
                ptr += sizeof(ushort);

                pkt.TopInfo = ProtocolVariable.GetTlsVersion(version) + " ";
                pkt.TopInfo += ProtocolVariable.GetTlsContentType(contentType) + " ";
                switch (contentType)
                {
                    case 22:
                        // Handshake, 读取handshake状态
                        byte handshakeType = *ptr;
                        ptr += 4; // 跳过length
                        pkt.TopInfo += ProtocolVariable.GetTlsHandshakeType(handshakeType) + " ";
                        break;
                }
            }
            else if (HeuristicJuderger.IsHttpProtocol(ptr))
            {
                // 是HTTP
                pkt.ApplicationLayerProtocol = ApplicationLayerProtocol.HTTP;
                pkt.TopProtocol = pkt.ApplicationLayerProtocol.ToString();
            }
            // 时间原因, 剩下协议就用端口暂时判断
            else if (pkt.TransmissionDestinationPort == 53 || pkt.TransmissionSourcePort == 53)
            {
                // DNS协议
                pkt.ApplicationLayerProtocol = ApplicationLayerProtocol.DNS;
                pkt.TopProtocol = pkt.ApplicationLayerProtocol.ToString();
                pkt.TopInfo = "DNS";
                DNSHeader dns = *(DNSHeader*)ptr;
                ushort identification = Tools.NetworkToHost(dns.identification);
                ushort type = Tools.NetworkToHost(dns.flags);
                if ((type & 0xf800) == 0x0000)
                {
                    pkt.TopInfo += String.Format(" Standard query 0x{0:X4}", identification);
                }
                else if ((type & 0xf800) == 0x8000)
                {
                    pkt.TopInfo += String.Format(" Standard query response 0x{0:X4}", identification);
                }
                // 详细内容就放在细粒度解析再说吧
            }
            break;
        case TransmissionLayerProtocol.UDP:
            // 解析UDP payload
    }
}

```

下面介绍简要启发式解析的相关内容。

TLS 流量中，能够通过定义知道，type 字段对应的字节一定在 20 到 23 之间，同时 version 字段对应的字节数一定在 0x0301 到 0x0304 之间，所以基于这种规则，编写了 TLS 流量判断器。

```

// 判断是否是TLS流量
1 个引用
public static unsafe bool IsTlsProtocol(byte* payloadPtr)
{
    byte type = (byte)(*payloadPtr);
    if (type >= 20 && type <= 23)
    {
        ushort* point = (ushort*)(payloadPtr + 1);
        int version = Tools.NetworkToHost(*point);
        if (version >= 0x0301 && version <= 0x0304)
            return true;
        else
            return false;
    }
    else
        return false;
}

```

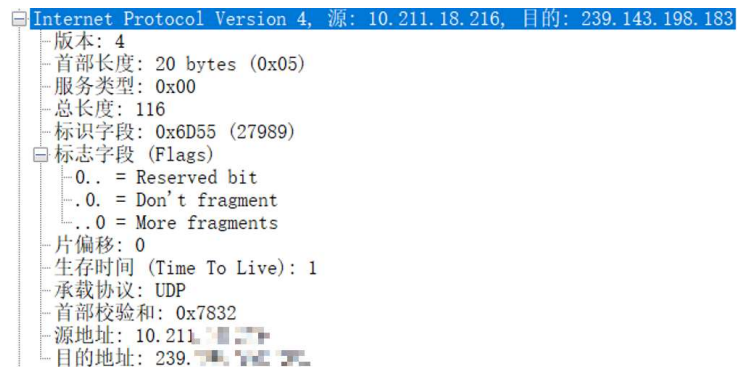
HTTP 流量也是类似的规则匹配，具体如代码和注释所示。

```
// 判断是否是HTTP流量
1 个引用
public static unsafe bool IsHttpProtocol(byte* payloadPtr)
{
    byte[] startArray = Tools.CopyFromPointer(payloadPtr, 0, 8);
    Tools.NetworkToHost(startArray);
    string res = Encoding.UTF8.GetString(startArray);
    // 判断前字符是否为HTTP/1.1, 为http response
    // 说明: 进一步可以设计启发式算法为"HTTP/1.1[20]状态码[20]状态描述[0x0d][0x0a]"来进一步提高置信度
    if (res.StartsWith("HTTP/1.1"))
    {
        return true;
    }

    // 如果非, 判断是否为四大HTTP操作 GET/POST/PUT/DELETE
    // 说明: 进一步可以设计启发式算法为"HTTP操作[0x20]URI[0x20]HTTP/1.1[0x0d][0x0a]"
    if (res.StartsWith("GET") || res.StartsWith("POST") || res.StartsWith("PUT") || res.StartsWith("DELETE"))
    {
        return true;
    }

    return false;
}
```

细粒度解析器主要被 MainForm 调用，其中增加了加载树状图的相关代码操作，以 IP 层为例，举例说明要加载如下的树状图，代码如何实现。



```
IPHeader ip = *(IPHeader*)ptr;
networkLayer = node.Add(string.Format("Internet Protocol Version 4, 源: {0}, 目的: {1}",
    pkt.NetworkLayerSource, pkt.NetworkLayerDestination));
networkLayer.Nodes.Add(string.Format("版本: 4"));
networkLayer.Nodes.Add(string.Format("首部长度: {0} bytes {0x{1:X2}}", pkt.NetworkLayerHeaderLength, (ip.versionAndHeadLength & 0x0F)));
networkLayer.Nodes.Add(string.Format("服务类型: 0x{0:X2}", ip.typeOfService));
networkLayer.Nodes.Add(string.Format("总长度: {0}", Tools.NetworkToHost(ip.totalLength)));
networkLayer.Nodes.Add(string.Format("标识字段: 0x{0:X4} ({0})", Tools.NetworkToHost(ip.identification)));
int df = (Tools.NetworkToHost(ip.flagAndOffset) & 0x4000) >> 14;
int mf = (Tools.NetworkToHost(ip.flagAndOffset) & 0x2000) >> 13;
TreeNode ipFlags;
if (df == 1)
{
    ipFlags = networkLayer.Nodes.Add(string.Format("标志字段 (Flags): 不要分片 Don't fragment"));
}
else if (mf == 1)
{
    ipFlags = networkLayer.Nodes.Add(string.Format("标志字段 (Flags): 更多分片 More fragments"));
}
else
{
    ipFlags = networkLayer.Nodes.Add(string.Format("标志字段 (Flags)"));
}
ipFlags.Nodes.Add(string.Format("{0}.. = Reserved bit",
    (Tools.NetworkToHost(ip.flagAndOffset) & 0x8000) >> 15));
ipFlags.Nodes.Add(string.Format(".{0}. = Don't fragment", df));
ipFlags.Nodes.Add(string.Format("..{0} = More fragments", mf));
networkLayer.Nodes.Add(string.Format("片偏移: {0}",
    (Tools.NetworkToHost(ip.flagAndOffset) & 0x1fff)));
networkLayer.Nodes.Add(string.Format("生存时间 (Time To Live): {0}", ip.ttl));
networkLayer.Nodes.Add(string.Format("承载协议: {0}", pkt.TransmissionLayerProtocol));
networkLayer.Nodes.Add(string.Format("首部校验和: 0x{0:X4}", Tools.NetworkToHost(ip.checksum)));
networkLayer.Nodes.Add(string.Format("源地址: {0}", pkt.NetworkLayerSource));
TransmissionLayer.Nodes.Add(string.Format("目的地址: {0}", pkt.NetworkLayerDestination));
TransmissionLayer(pkt, ptr + pkt.NetworkLayerHeaderLength, node);
break;
```

同时，MainForm 中为了展示详细信息需要展示包的 16 进制原始信息，包含行号、16 进制信息和可显示字符显示。具体实现方法如下。

```

public string FormatByteArray(byte[] byteArray)
{
    int length = byteArray.Length;
    StringBuilder lineNumberBuilder = new StringBuilder();
    StringBuilder hexBuilder = new StringBuilder();
    StringBuilder charBuilder = new StringBuilder();

    int lineCounter = 0;

    for (int i = 0; i < length; i++)
    {
        hexBuilder.AppendFormat("{0:X2} ", byteArray[i]);
        charBuilder.Append(byteArray[i] >= 32 && byteArray[i] <= 126 ? (char)byteArray[i] : '.'); // 可打印字符显示为字符, 其他显示为'.'

        // 在Wireshark样式的输出中, 每16个字节分为一组, 以空格分隔
        if ((i + 1) % 16 == 0 || i == length - 1)
        {
            lineNumberBuilder.AppendFormat("{0:X4} ", lineCounter);
            hexBuilder.Append(" "); // 每16个字节后加两个空格分隔
            if (i == length - 1)
            {
                // 最后一个字节, 需要补齐hex这一行的空格, 否则ASCII会错位
                hexBuilder.Append(" ".PadRight((16 - (length % 16)) * 3));
            }
            lineNumberBuilder.Append(charBuilder.ToString());
            lineNumberBuilder.AppendLine(hexBuilder.ToString());
            hexBuilder.Clear();
            charBuilder.Clear();
            lineCounter += 16;
        }
    }

    return lineNumberBuilder.ToString();
}

```

实验遇到的问题与实验总结

1、抓包效率、解析效率与显示效率问题

就像本实验报告中“软件关键内容梳理”部分中提到的那样, 通过多线程、“懒加载”技术和 unsafe 关键字提供的指针操作能力, 可以大幅度提升本软件的各种效率。作为对比, 使用这些措施之前, 本软件在校园网环境内使用千兆网卡在公网跑测速时, 会导致软件卡死、Buffer 溢出等问题, 进行优化之后没有再出现崩溃、卡死的问题。

2、大端序和小端序问题

大小端存储的划分是为了解决长度大于一个字节的类型内容在存储地址上以不同顺序分布的问题。若最高有效位存于最低内存地址处, 最低有效位存于最高内存处, 这样的存储方式称为“大端字节序”, 反之为“小端字节序”。

网络协议规定, 把接收到的第一个字节当作高位字节看待, 也就是说网络字节序是大端序, 而通常情况下个人计算机上的数据都以小端存储。这时, 就需要对字节顺序进行转换, 否则会出现错误。本软件中通过位运算实现了大小端序的转换函数, 并应用在包解析过程中。

```

public static ushort NetworkToHost(ushort value)
{
    if (BitConverter.IsLittleEndian)
        return (ushort)((value >> 8) | (value << 8));
    else
        return value;
}

```


3、16 进制显示问题

在进行 16 进制的显示时，一开始通过直接显示 16 进制对应的字符会出现错误。后来发现有许多字节对应的都是不可显示字符，故加入了判断机制，用“.”字符代替不可显示字符。具体实现代码在上文“协议解析具体设计”部分有所描述。

4、思考

(2) 如何识别某种特定类型的应用层报文？

在解析 TCP 和 UDP 协议的 payload 部分中，有一个关键问题就是如何判断 payload 是哪些应用层的协议。不像底层协议会明确有字段规定上层使用哪个协议，传输层协议并无这样的字段。在早期时代，一般通过端口的形式判断，但本质上来说网络协议并没有明确规定每个服务必须使用哪些端口，端口仅仅用来区分主机上的进程。现代网络中更是如此（如 HTTP 服务可以开启在 8080 等端口上）。为解决这个问题，特借鉴“启发式解析”的相关内容，即根据 payload 的特征进行规则上的匹配，进而针对该协议进行具体解析。本软件中简要实现的启发式解析匹配规则在“HeuristicJudger.cs”中有所体现。通过阅读源码和查找资料，业界软件 Wireshark 在判断 payload 协议特征时，也是基于这种模式进行判断和解析应用层 payload。

（ 参 考 链 接 ： <https://www.wireshark.org/lists/wireshark-dev/200808/msg00252.html> ）

5、实验总结

通过本次实验，我实现了一个简单的网络数据包嗅探与解析工具。

在实验过程中，我也遇到了许多问题，参考了许多前人的工作。个人认为本软件还有许多可以改进的地方，但由于时间和任务量的关系，无法把这个软件做的尽善尽美。

我在以前使用 Wireshark 时，只是知其然而不知其所以然。通过实现本软件，我对网络工作的原理有了更进一步的理解，掌握了网络嗅探器的基本工作原理，体会了网络协议设计中的众多精髓，为以后的学习和研究打下坚实的基础。