

# 第二次实验：IP 数据报捕获与分析

1711342 李纪

2019 年 11 月 2 日

## 摘要

这是我的**第二次实验**的实验报告，请老师查阅，谢谢老师。

**关键字：**MFC、WinPcap、IP 数据报

# 目录

<b>1 实验的目的</b>	<b>3</b>
<b>2 设计思路</b>	<b>3</b>
2.1 利用 WinPcap 捕获数据包 . . . . .	3
2.1.1 获取设备列表 . . . . .	3
2.1.2 打开网络接口 . . . . .	4
2.1.3 在打开的网络接口卡上捕获网络数据包 . . . . .	5
2.2 MFC 实现 . . . . .	5
<b>3 实验过程</b>	<b>5</b>
<b>4 关键部分源代码</b>	<b>6</b>
4.1 头文件（函数和变量的声明） . . . . .	6
4.2 功能：检测出所有的网络接口卡 . . . . .	7
4.3 功能：显示设备详细信息 . . . . .	9
4.4 功能：捕获数据报 . . . . .	10
4.4.1 子功能：计算校验和，显示结果 . . . . .	13
<b>5 程序演示</b>	<b>18</b>
5.1 程序界面介绍 . . . . .	18
5.2 程序逻辑介绍 . . . . .	18
<b>6 实验现象和结论的分析（验证程序正确性）</b>	<b>19</b>
<b>A IPv4 header checksum</b>	<b>22</b>
A.1 Computation . . . . .	22
A.2 Examples . . . . .	22
A.2.1 Calculating the IPv4 header checksum . . . . .	22
A.2.2 Verifying the IPv4 header checksum . . . . .	23

## 1 实验的目的

在对网络的安全性和可靠性进行分析时，网络管理员通常需要对网络中传输的数据包进行监听和分析。目前，Internet 中流行的数据包监听与分析工具很多（如 snort, iris, tcpdump 等），但本实验要求通过 WinPcap（或 LibPcap）编制一个简单的 IP 网络数据报捕获与分析程序，学习 IP 数据报校验和计算方法，初步掌握网络监听与分析技术的实现过程，加深对网络协议的理解。

## 2 设计思路

设计思路主要参考了《网络技术与应用》[2] 书上的内容。

### 2.1 利用 WinPcap 捕获数据包

WinPcap 是一个开源的数据包捕获体系架构，它的主要功能是进行数据包捕获和网络分析。inPcap 包括了内核级别的包过滤、低层次的动态链接库（packet.dll）高级别系统无关的函数库（wpcap.dll）等。详细信息请参见 <http://www.winpcap.org>。本实验将利用 WinPcap 高级别系统无关函数库中提供的函数对流经网卡的数据包进行捕获。

利用 WinPcap 捕获数据包一般需要经过下面 3 个步骤。

#### 2.1.1 获取设备列表

在开发以 WinPcap 为基础的应用程序时，第一步需要获取网络接口设备（网卡）列表。获取网络接口设备列表可以调用 WinPcap 提供的 pcap\_findalldevs\_ex() 函数，该函数的原型如下。

```
1 int pcap_findalldevs_ex(  
2     const char *source, // 指定从哪里获取网络接口列表。  
3     struct pcap_rmtauth *auth, // 该参数对获取本机的网络接口列表没有任何意义，  
        设值为 NULL 即可。  
4     pcap_if_t **alldevs, // 本函数返回后，alldevs 参数指向获取的网络接口列表的  
        第一个元素。  
5     char *errbuf // 用户定义的存放错误信息的缓冲区。  
6 );
```

调用发生错误时，pcap\_findalldevs\_ex() 返回-1，具体的错误信息可以从 errbuf 参数中获得。调用成功时，pcap\_findalldevs\_ex() 返回 0，这时 alldevs 参数指向网络接口链表的第一个元素。

在 `alldevs` 指向的网络接口链表中，每一个元素都是一个 `pcap_if_t` 结构。`pcap_if_t` 的定义如下。

```
1 typedef struct pcap_if pcap_if_t;
2
3 struct pcap_if {
4     struct pcap_if *next;
5     char *name; /* name to hand to "pcap_open_live()" */
6     char *description; /* textual description of interface, or NULL */
7     struct pcap_addr *addresses;
8     bpf_u_int32 flags; /* PCAP_IF_ interface flags */
9 };
```

在使用 `pcap_findalldevs_ex()` 返回网络接口设备列表后，可以使用 `pcap_freealldevs()` 释放该设备列表。`pcap_freealldevs()` 函数的原型如下。

```
1 void pcap_freealldevs(pcap_if_t *alldevsp);
```

其中, `alldevsp` 指向需要释放的设备链表的第一个元素, `pcap_freealldevs()` 函数返回。

### 2.1.2 打开网络接口

得到网络接口设备列表之后，可以选择感兴趣的网络接口卡并对其上的网络流量进行监听。在对某一网络接口卡进行监听之前，首先需要将其打开。打开某一网络接口设备可以使用 WinPcap 提供的 `pcap_open()` 函数。`pcap_open()` 函数的原型如下。

```
1 pcap_t *pcap_open(
2     const char *source, // 需要打开的网卡的名字
3     int snaplen, // WinPcap 获取网络数据包的最大长度。
4     int flags, // 指定以何种方式打开网络接口设备并获取网络数据包。
5     int read_timeout, // 数据包捕获函数等待一个数据包的最大时间
6     struct pcap_rmtauth *auth, // 在远程设备中捕获网络数据包时使用。在编写捕获
    // 本机网络数据包的应用程序中，需要将 auth 设置为 NULL
7     char *errbuf // 用户定义的存放错误信息的缓冲区。
8 );
```

调用出错时, `pcap_open()` 函数返回 `NULL`，可以通过 `errbuf` 获取错误的详细信息。如果调用成功，则 `pcap_open()` 返回一个指向 `pcap_t` 的指针，该指针将在后续调用的函数（如 `pcap_next_ex()` 等）中使用。

### 2.1.3 在打开的网络接口卡上捕获网络数据包

一旦打开网络接口卡，就可以利用 WinPcap 提供的函数捕获流经的网络数据包 WinPcap 提供了多种不同的方法捕获数据包,其中,pcap\_dispatch (和 pcap\_loop()通过回调函数将捕获的数据包传递给应用程序,而 pcap\_next\_ex() 则不使用回调函数。下面以 pcap\_next\_ex() 为例介绍数据包的捕获过程。

WinPcap 提供的 pcap\_next\_ex() 的函数原型如下。

```
1 int    pcap_next_ex(  
2     pcap_t *, // 通过该参数指定捕获哪块网卡上的网络数据包。  
3     struct pcap_pkthdr **, // 在 pcap_next_ex() 函数调用成功后,  
4         // 该参数指向的 pcap_pkthdr 结构保存有所捕获网络数据包的一些基本  
5         信息  
6     const u_char ** // pkt_data: 指向捕获到的网络数据包。  
7 );
```

调用 pcap\_next\_ex() 函数可能返回 1、0、-1 等不同的值。如果 pcap\_next\_ex() 函数正确捕获到一个数据包，那么，它将返回 1。这时，pkt\_header 保存有捕获数据包的一些基本信息，而 pkt\_data 指向捕获数据包的完整数据。如果在 pcap\_open() 函数中指定的时间范围内 (read\_timeout) 没有捕获到任何网络数据包，那么 pcap\_next\_ex() 函数将返回 0。尽管这不是一种错误，但 pkt\_header 和 pkt\_data 参数都不可用。如果在调用过程中发生错误，那么 pcap\_next\_ex() 函数将返回-1。

## 2.2 MFC 实现

按照书上 6.5.3 部分内容正确配置 MFC 开发环境。[2]

源代码讲解在**关键部分源代码**部分。此处略过。

## 3 实验过程

先理解实验中涉及到的原理（比如计算头部校验和），然后根据书中内容正确配置环境，紧接着写入关键部分的源代码。

计算头部校验和的原理在附录 A 中解释，本实验只涉及到 IPv4 的头部校验和计算。

## 4 关键部分源代码

(“...” 部分表示省略)

此部分分为 4 个小节，第一个小节展示函数和变量的声明，后三个小节每个小节解释一个具体功能的实现。每小节开头都有**代码摘要**，粗略地展示了功能的实现步骤。

代码解释大部分可以根据代码里的注释来解释说明，请老师查阅。为了方便解释与理解，代码之间的联系与具体实现请直接通过 Visual Studio 查看源代码。

### 4.1 头文件（函数和变量的声明）

**代码摘要：**函数和变量的声明。

因为很多变量都在头文件中声明，所以为了便于理解，首先贴上对话框类头文件（CapturePacketDlg.h）的代码。

```
1      ...
2  class CCapturePacketDlg : public CDialogEx
3  {
4      ...
5  public:
6      pcap_if_t* alldevs = nullptr; // 指向设备链表首部的指针
7      pcap_if_t* curdev = nullptr; // 一个全局指针，指向当前选中的设备
8      pcap_t* adhandle = nullptr; // 一个句柄
9      CWinThread* m_Capturer = nullptr; // 启动工作者线程的指针
10     char errbuf[PCAP_ERRBUF_SIZE]; // 错误信息缓冲区
11     bool m_capStatus = false; // 判断是否处于捕获状态
12     void displayString(CEdit& editCtrl, CString& str); // 在编辑控件中显示内
        容
13     afx_msg void OnBnClickedButtoncapture();
14
15     // 捕获报文按钮
16     CButton m_capture;
17     // 返回按钮
18     CButton m_return;
19     // 停止捕获按钮
20     CButton m_stopCapture;
21     // 过滤器控件
22     CEdit m_filter;
23     // 日志控件
24     CEdit m_log;
25     // 网络接口列表
26     CListBox m_list_interface;
27     // 网络接口信息列表
28     CListBox m_list_interfaceInfo;
```

```

29  afx_msg void OnSelchangeListinterface();
30  void UpdateInfo(); // 更新捕获接口的详细信息框
31  afx_msg void OnBnClickedButtonstopcapture();
32  afx_msg void OnClose();
33  afx_msg void OnBnClickedButtonreturn();
34 };
35
36 // 全局函数
37 UINT Capturer(PVOID hWnd); // 数据包捕获工作者线程的控制函数
38
39 #pragma pack(1) // 进入字节对齐方式
40 typedef struct FrameHeader_t { // 帧首部
41     BYTE  DesMAC[6]; // 目的地址
42     BYTE  SrcMAC[6]; // 源地址
43     WORD  FrameType; // 帧类型
44 } FrameHeader_t;
45
46 typedef struct IPHeader_t { // IP 首部
47     BYTE  Ver_HLen;
48     BYTE  TOS;
49     WORD  TotallLen;
50     WORD  ID;
51     WORD  Flag_Segment;
52     BYTE  TTL;
53     BYTE  Protocol;
54     WORD  Checksum;
55     ULONG SrcIP;
56     ULONG DstIP;
57     WORD  Opt[20];
58 } IPHeader_t;
59
60 typedef struct Data_t { // 包含帧首部和 IP 首部的数据包
61     FrameHeader_t FrameHeader;
62     IPHeader_t     IPHeader;
63 } Data_t;
64
65 #pragma pack() // 恢复缺省对齐方式

```

Listing 1: CapturePacketDlg.h

## 4.2 功能：检测出所有的网络接口卡

**代码摘要：**在对话框初始化函数 CCapturePacketDlg::OnInitDialog() 中查找设备。

### 1. CCapturePacketDlg::OnInitDialog()

- 获取网络接口卡设备列表
- 在列表框控件中打印设备列表

- 完成界面初始化（默认选择第一个设备，显示第一个设备的详情；禁用“停止捕获”控件）

```

1 BOOL CCapturePacketDlg::OnInitDialog()
2 {
3     ...
4     int i = 0; // 标识找到 i 个设备
5
6     /* Retrieve the device list */
7     if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL, &alldevs, errbuf) == -1)
8     {
9         fprintf(stderr, "Error in pcap_findalldevs: %s\n", errbuf);
10        AfxMessageBox((CString)errbuf);
11        exit(1);
12    }
13
14    /* Print the list */
15    for (pcap_if_t* d = alldevs; d; d = d->next)
16        m_list_interface.InsertString(-1, (CString)("%d. %s\r\n", ++i, d->name))
17        ;
18    UpdateData(true);
19    Invalidate(true);
20    UpdateWindow(); // 更新窗口
21
22    if (i == 0) // 如果没有检测到设备
23    {
24        AfxMessageBox(_T("No interfaces found! Make sure Npcap is installed."));
25        return -1;
26    }
27
28    m_list_interface.SetCurSel(0); // 默认选中第一行
29    int cur = m_list_interface.GetCurSel(); // 获取listbox被选中的行的数目
30
31    // 找到当前指向的设备
32    curdev = alldevs;
33    while (cur--)
34        curdev = curdev->next;
35    UpdateInfo();
36
37    // 禁用“停止捕获”控件
38    m_stopCapture.EnableWindow(false);
39
40    return TRUE; // 除非将焦点设置到控件，否则返回 TRUE
41 }

```

Listing 2: OnInitDialog()



### 4.3 功能：显示设备详细信息

**代码摘要：**显示设备详细信息结合了两个自定义函数(OnSelchangeListinterface() 和 UpdateInfo()) 进行实现。

#### 1. CCapturePacketDlg::OnSelchangeListinterface()

- 找到当前指向的设备
- 调用 UpdateInfo() 刷新窗口

#### 2. CCapturePacketDlg::UpdateInfo()

- 更新捕获接口的详细信息 (IP 地址、广播信息、网络掩码、目的地址)

```
1 void CCapturePacketDlg::OnSelchangeListinterface()
2 {
3     int cur = m_list_interface.GetCurSel(); // 获取listbox被选中的行的数目
4
5     // 找到当前指向的设备
6     curdev = alldevs;
7     while (cur--)
8         curdev = curdev->next;
9     UpdateInfo();
10 }
11
12 void CCapturePacketDlg::UpdateInfo()
13 {
14     // 更新捕获接口的详细信息
15     m_list_interfaceInfo.ResetContent(); // 清除原有框的内容
16     m_list_interfaceInfo.InsertString(-1, (CString(curdev->name))); // 显
17     // 示该网络接口设备的名字
18     m_list_interfaceInfo.InsertString(-1, (CString(curdev->description))); //
19     // 显示该网络接口设备的描述信息
20
21     sockaddr_in* temp;
22     IN_ADDR temp1;
23     char* temp_data;
24     CString output;
25     for (pcap_addr* a = curdev->addresses; a != NULL; a = a->next)
26     {
27         if (a->addr->sa_family == AF_INET) // 判断地址是否为 IP 地址
28         {
29             temp = (sockaddr_in*)(a->addr);
30             temp1 = temp->sin_addr;
31             temp_data = inet_ntoa(temp1);
32             output = _T("IP address: ") + (CString(temp_data)); // 获取 IP 地址
```

```

31     m_list_interfaceInfo.InsertString(-1, output);
32
33     temp = (sockaddr_in*)(a->netmask);
34     temp1 = temp->sin_addr;
35     temp_data = inet_ntoa(temp1);
36     output = _T("Netmask: ") + (CString(temp_data)); // 获取网络掩码
37     m_list_interfaceInfo.InsertString(-1, output);
38
39     temp = (sockaddr_in*)(a->broadaddr);
40     temp1 = temp->sin_addr;
41     temp_data = inet_ntoa(temp1);
42     output = _T("Broadcast address: ") + (CString(temp_data)); // 获取广播
地址
43     m_list_interfaceInfo.InsertString(-1, output);
44
45     temp = (sockaddr_in*)(a->dstaddr);
46     if (temp != nullptr)
47     {
48         temp1 = temp->sin_addr;
49         temp_data = inet_ntoa(temp1);
50         output = _T("Destination address: ") + (CString(temp_data)); // 获取
目的地址
51         m_list_interfaceInfo.InsertString(-1, output);
52     }
53 }
54 }
55
56 return void();
57 }

```

Listing 3: OnSelchangeListinterface()、UpdateInfo()

#### 4.4 功能：捕获数据报

**代码摘要：**捕获数据报结合了两个自定义函数(OnBnClickedButtoncapture() 和 Capturer(PVOID hWnd)) 进行实现。

##### 1. Capturer(PVOID hWnd)

- 找到当前指向的设备
- 打开网络接口卡
- 在打开的网络接口卡上捕获网络数据包

##### 2. CCApturePacketDlg::OnBnClickedButtoncapture()

- 调用 AfxBeginThread() 函数启动工作者线程

```

1  UINT Capturer(PVOID hWnd) // 数据包捕获工作者线程的控制函数
2  {
3      CCapturePacketDlg* dlg = (CCapturePacketDlg*)theApp.m_pMainWnd; // 获取对话框句柄
4
5      int cur = dlg->m_list_interface.GetCurSel(); // 获取listbox被选中的行的数目
6      dlg->curdev = dlg->alldevs;
7      while (cur--)
8          dlg->curdev = dlg->curdev->next;
9
10     char* errbuf = new char[100];
11
12     // 在对某一网络接口卡进行监听之前，首先需要将其打开。打开某一网络接口设备
        可以使用 WinPcap 提供的 pcap_open() 函数
13     dlg->adhandle = pcap_open(
14         dlg->curdev->name // 需要打开的网卡的名字
15         , 65536 // WinPcap 获取网络数据包的最大长度。设为 2^16
16         , PCAP_OPENFLAG_PROMISCUOUS // 它通知系统以混杂模式打开网络接口设备。
17         , 1000 // 数据包捕获函数等待一个数据包的最大时间，设为 1 秒
18         , NULL // 在远程设备中捕获网络数据包时使用。在编写捕获本机网络数据包的应用
        程序中，需要将 auth 设置为 NULL
19         , errbuf // 用户定义的存放错误信息的缓冲区。
20     );
21
22     // 调用出错时，pcap_open() 函数返回 NULL，可以通过 errbuf 获取错误的详细信息
23     if (dlg->adhandle == NULL)
24     {
25         AfxMessageBox(_T("打开网卡出错：") + (CString)errbuf);
26         return -1;
27     }
28
29
30     while (dlg->m_capStatus == true)
31     {
32         // 在打开的网络接口卡上捕获网络数据包
33         int pkt = pcap_next_ex(
34             dlg->adhandle, // pcap_next_ex() 函数通过该参数指定捕获哪块网卡上的网
            络数据包。
35             &pkt_header, // 在 pcap_next_ex() 函数调用成功后，
36             // 该参数指向的 pcap_pkthdr 结构保存有所捕获网络数据包的一些基本
            信息
37             &pkt_data // pkt_data：指向捕获到的网络数据包。
38         );
39
40         // 如果在调用过程中发生错误，那么 pcap_next_ex() 函数将返回 -1
41         if (pkt == -1)
42         {
43             AfxMessageBox(_T("在调用过程中发生错误，pcap_next_ex() 函数返回 -1"));
44             return -2;
45         }

```

```

46 // 指定的时间范围内 (read_timeout) 没有捕获到任何网络数据包, 那么
    pcap_next_ex() 函数将返回 0
47 else if (pkt == 0)
48 {
49     // AfxMessageBox(_T("指定的时间范围内 (read_timeout) 没有捕获到任何网
    络数据包, pcap_next_ex() 函数返回 0"));
50     // return -3;
51     continue;
52 }
53 // 如果 pcap_next_ex() 函数, 正确捕获到一个数据包, 那么, 它将返回 1。
54 else if (pkt == 1)
55 {
56
57     dlg->SendMessage(WM_PACKET, 0, 0); // SendMessage 为同步式的消息发送,
58     // 它将消息放入窗口的消息队列后等待消息被处理后返回
59 }
60 }
61
62
63 return 0;
64 }
65
66 void CCapturePacketDlg::OnBnClickedButtoncapture()
67 {
68     m_capStatus = true;
69     m_capture.EnableWindow(false);
70     m_stopCapture.EnableWindow(true);
71
72     // 调用 AfxBeginThread() 函数启动工作者线程
73     // AfxBeginThread() 函数将返回一个指向新创建线程对象的指针
74     m_Capturer = AfxBeginThread(
75         (AFX_THREADPROC)Capturer, // pfnThreadProc: 指向工作者线程的控制函数, 它
        的值不能为 NULL
76         NULL, //
77         THREAD_PRIORITY_NORMAL // 用于指定线程的优先级
78     );
79
80     if (m_Capturer == NULL) {
81         AfxMessageBox(L"启动捕获数据包线程失败!", MB_OK | MB_ICONERROR);
82         return;
83     }
84     else /*打开选择的网卡 */
85     {
86         CString temp = _T("监听 ") + (CString(curdev->description)) + _T("\r\n")
87         ;
88         displayString(m_log, temp + bar);
89     }
90     UpdateData(true);
91     Invalidate(true);
92     UpdateWindow();

```

Listing 4: OnBnClickedButtoncapture()、Capturer(PVOID hWnd)

#### 4.4.1 子功能：计算校验和，显示结果

**代码摘要：**计算校验和，显示结果部分结合了三个自定义函数(OnPacket(WPARAM wParam, LPARAM lParam)、IpChecksum(IPHeader\_t IPHeader) 和 CheckOverflow(DWORD& now)) 进行实现。

1. CCapturePacketDlg::OnPacket(WPARAM wParam, LPARAM lParam)
  - 过滤掉所有非 IPv4 的数据报
  - 显示相关数据
  - 调用 IpChecksum(IPHeader\_t IPHeader) 计算头部校验和
2. IpChecksum(IPHeader\_t IPHeader)
  - 根据算法（参考附录 A）计算头部校验和
3. CheckOverflow(DWORD& now)
  - 辅助 IpChecksum(IPHeader\_t IPHeader) 函数，实现回卷

```

1 // 定义一个前十六位全为 0，后十六位全为 1 的常量
2 const DWORD overflowCrush = 0b00000000000000001111111111111111;
3 WORD CheckOverflow(DWORD& now)
4 {
5     // 如果现在的数比 max(WORD) 大了，说明溢出了
6     if (now > WORD_MAX)
7     {
8         // 回卷
9         now = overflowCrush & now;
10        now++;
11    }
12    return now;
13 }
14
15 WORD IpChecksum(IPHeader_t IPHeader)
16 {
17     // 考虑可能溢出，采用 DWORD 类型
18     DWORD CheckSum, temp1, temp2;
19
20     // 按照算法一步一步做
21     temp1 = (IPHeader.TOS << 8) + IPHeader.Ver_HLen;

```

```

22 temp2 = IPHeader.TotalLen;
23 CheckSum = temp1 + temp2;
24 CheckOverflow(CheckSum);
25
26 temp1 = IPHeader.ID;
27 temp2 = CheckSum;
28 CheckSum = temp1 + temp2;
29 CheckOverflow(CheckSum);
30
31 temp1 = IPHeader.Flag_Segment;
32 temp2 = CheckSum;
33 CheckSum = temp1 + temp2;
34 CheckOverflow(CheckSum);
35
36 temp1 = (IPHeader.Protocol << 8) + IPHeader.TTL;
37 temp2 = CheckSum;
38 CheckSum = temp1 + temp2;
39 CheckOverflow(CheckSum);
40
41 // IP 数据报的头部校验和就不加了
42
43 temp1 = IPHeader.SrcIP >> 16;
44 temp2 = CheckSum;
45 CheckSum = temp1 + temp2;
46 CheckOverflow(CheckSum);
47
48 temp1 = IPHeader.SrcIP & overflowCrush;
49 temp2 = CheckSum;
50 CheckSum = temp1 + temp2;
51 CheckOverflow(CheckSum);
52
53 temp1 = IPHeader.DstIP >> 16;
54 temp2 = CheckSum;
55 CheckSum = temp1 + temp2;
56 CheckOverflow(CheckSum);
57
58 temp1 = IPHeader.DstIP & overflowCrush;
59 temp2 = CheckSum;
60 CheckSum = temp1 + temp2;
61 CheckOverflow(CheckSum);
62
63 // 考虑可选和填充部分
64 int HLen = IPHeader.Ver_HLen & 0x0f;
65 int OptLen = HLen - 5;
66
67 for (int i = 0; i < OptLen; ++i)
68 {
69     temp1 = IPHeader.Opt[i];
70     temp2 = CheckSum;
71     CheckSum = temp1 + temp2;
72     CheckOverflow(CheckSum);

```

```

73     }
74
75     return (~((WORD)Checksum));
76 }
77
78 // 消息处理函数
79 LRESULT CCapturePacketDlg::OnPacket(WPARAM wParam, LPARAM lParam)
80 {
81     // 处理捕获到的数据包
82
83     Data_t* IPacket;
84     WORD RecvChecksum; // 头部校验和
85     WORD Identifier; // 标识
86     WORD CheckSum; // 计算出的头部校验和
87
88
89     IPacket = (Data_t*)pkt_data;
90     WORD FrameType = (IPacket->FrameHeader.FrameType);
91     BYTE Ver_HLen = (IPacket->IPHeader.Ver_HLen >> 4);
92
93
94
95     // IPv6 没有头部校验和，下面是维基百科的说法
96     // In order to increase performance,
97     // and since current link layer technology and transport or application
98     // layer protocols are assumed to provide sufficient error detection,
99     // the header has no checksum to protect it.
100
101     if (FrameType == 8 && Ver_HLen == 4) // 保证是 IP 数据报，保证版本为 IPv4
102     // if (FrameType == 2048) // 保证是 IP 数据报，保证版本正确
103     {
104         Identifier = IPacket->IPHeader.ID;
105         RecvChecksum = IPacket->IPHeader.Checksum;
106         CheckSum = IpChecksum(IPacket->IPHeader);
107
108         // 下面是输出
109         // 时间输出
110         CString timeStr;
111         timeStr.Format(_T(" %d "), pkt_header->ts.tv_sec);
112         time_t t = pkt_header->ts.tv_sec;
113         tm* timeptr = localtime(&t);
114         char buffer[80];
115         strftime(buffer, sizeof(buffer), "%Y/%m/%d %H:%M:%S", timeptr);
116         timeStr.Format(_T("%s.%d"), (CString)buffer, pkt_header->ts.tv_usec);
117
118         // IP 报文头长度输出
119         CString len;
120         // len.Format(_T(" len: %d FrameType: %d\r\n"), pkt_header->caplen,
121         // FrameType);
122         // len.Format(_T(" len: %d Ver_HLen: %d\r\n"), pkt_header->caplen,
123         // Ver_HLen);

```

```

121     len.Format(_T(" len: %d \r\n"), IPPacket->IPHeader.TotalLen);
122     displayString(m_log, timeStr + len);
123
124     // 标识：头部校验和： 计算出的头部校验和：
125     CString info;
126     info.Format(_T("标识： %04x  头部校验和：  %04x  计算出的头部校验和： %04x\r\n"),
127         ntohs(Identifier), ntohs(RecvChecksum), ntohs(CheckSum));
128     //info.Format(_T("标识： %04x  头部校验和：  %04x  计算出的头部校验和： %04x\r\n"),
129     //    (Identifier), (RecvChecksum), (CheckSum));
130     displayString(m_log, info + bar);
131 }
132
133
134 return LRESULT();
135 }
136
137
138 // error C2440: “类型转换” : 无法从“overloaded-function”转换为 xxx
139 // 原因：线程函数需要静态成员函数或全局函数
140
141 UINT Capturer(PVOID hWnd) // 数据包捕获工作者线程的控制函数
142 {
143     CCapturePacketDlg* dlg = (CCapturePacketDlg*)theApp.m_pMainWnd; //获取对话框句柄
144
145     int cur = dlg->m_list_interface.GetCurSel(); // 获取listbox被选中的行的数目
146     dlg->curdev = dlg->alldevs;
147     while (cur--)
148         dlg->curdev = dlg->curdev->next;
149
150     char* errbuf = new char[100];
151
152     // 在对某一网络接口卡进行监听之前，首先需要将其打开。打开某一网络接口设备
153     // 可以使用 WinPcap 提供的 pcap_open() 函数
154     dlg->adhandle = pcap_open(
155         dlg->curdev->name // 需要打开的网卡的名字
156         , 65536 // WinPcap 获取网络数据包的最大长度。设为 2^16
157         , PCAP_OPENFLAG_PROMISCUOUS // 它通知系统以混杂模式打开网络接口设备。
158         , 1000 // 数据包捕获函数等待一个数据包的最大时间，设为 1 秒
159         , NULL // 在远程设备中捕获网络数据包时使用。在编写捕获本机网络数据包的应用
160         // 程序中，需要将 auth 设置为 NULL
161         , errbuf // 用户定义的存放错误信息的缓冲区。
162     );
163
164     // 调用出错时，pcap_open() 函数返回 NULL，可以通过 errbuf 获取错误的详细信息
165     if (dlg->adhandle == NULL)
166     {
167         AfxMessageBox(_T("打开网卡出错：") + (CString)errbuf);
168     }
169 }

```



```

166     return -1;
167 }
168
169
170 while (dlg->m_capStatus == true)
171 {
172     // 在打开的网络接口卡上捕获网络数据包
173     int pkt = pcap_next_ex(
174         dlg->adhandle, // pcap_next_ex() 函数通过该参数指定捕获哪块网卡上的网
        络数据包。
175         &pkt_header, // 在 pcap_next_ex() 函数调用成功后,
176         // 该参数指向的 pcap_pkthdr 结构保存有所捕获网络数据包的一些基本
        信息
177         &pkt_data // pkt_data: 指向捕获到的网络数据包。
178     );
179
180     // 如果在调用过程中发生错误, 那么 pcap_next_ex() 函数将返回 -1
181     if (pkt == -1)
182     {
183         AfxMessageBox(_T("在调用过程中发生错误, pcap_next_ex() 函数返回 -1"));
184         return -2;
185     }
186     // 指定的时间范围内 (read_timeout) 没有捕获到任何网络数据包, 那么
    pcap_next_ex() 函数将返回 0
187     else if (pkt == 0)
188     {
189         // AfxMessageBox(_T("指定的时间范围内 (read_timeout) 没有捕获到任何网
        络数据包, pcap_next_ex() 函数返回 0"));
190         // return -3;
191         continue;
192     }
193     // 如果 pcap_next_ex() 函数, 正确捕获到一个数据包, 那么, 它将返回 1。
194     else if (pkt == 1)
195     {
196
197         dlg->SendMessage(WM_PACKET, 0, 0); // SendMessage 为同步式的消息发送,
        // 它将消息放入窗口的消息队列后等待消息被处理后返回
198
199     }
200 }
201
202
203 return 0;
204 }

```

Listing 5: IpChecksum(IPHeader\_t IPHeader)、CheckOverflow(DWORD&now)、OnPacket(WPARAM wParam LPARAM lParam)

## 5 程序演示

### 5.1 程序界面介绍

程序分为 3 个部分，分别是网络接口卡显示区、结果显示区以及控制区。

其中，网络接口卡显示区由程序上方的两个列表框控件组成，结果显示区由一个编辑框控件组成，控制区由三个按钮控件组成。中间的过滤条件未实现，故无效。

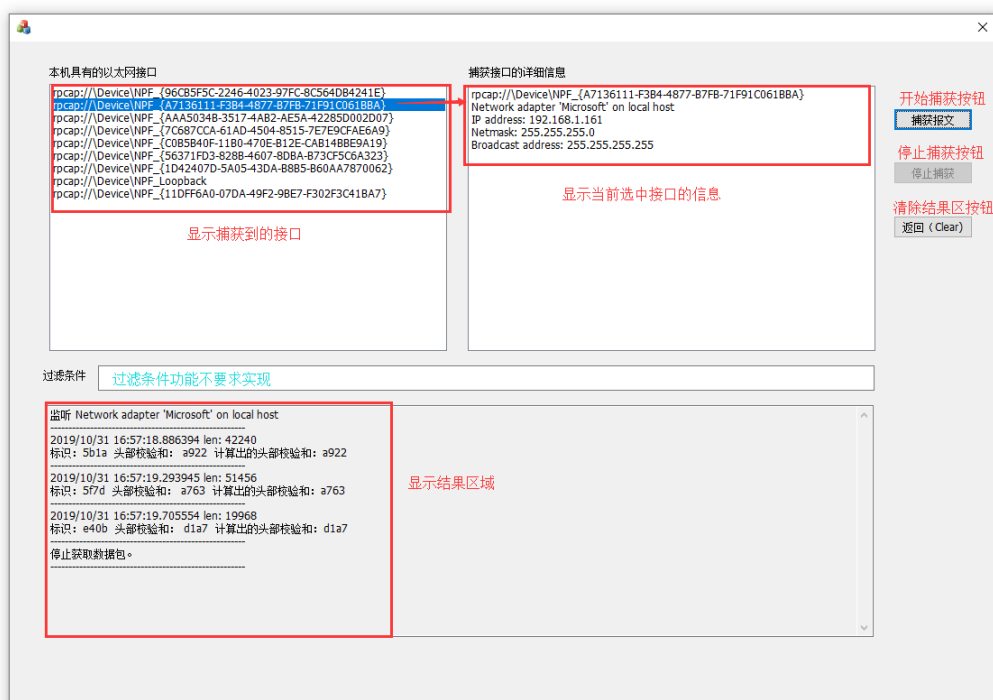


图 1: 程序界面介绍

### 5.2 程序逻辑介绍

1. 在左部的列表框控件中点击任意网络接口卡后，右部的列表框控件中会显示该卡相关信息，包括名字、IP 地址、网络掩码、广播地址等。
2. 点击“捕获报文”按钮后，程序会自动捕获通过当时选中的网络接口卡的 IPv4 报文。
3. 在下方的编辑框控件中显示 IPv4 报文相关信息，包括标识、(数据报

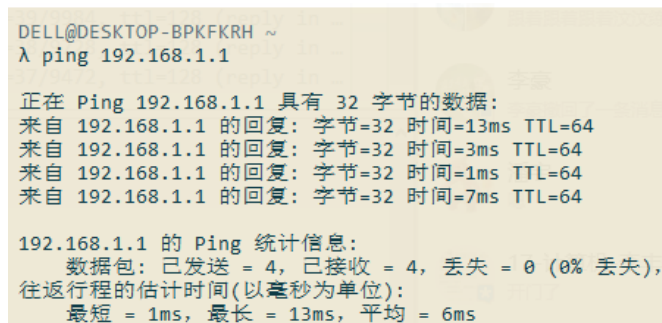
中的) 头部校验和、(程序) 计算出的头部校验和等。

4. 点击“停止捕获”按钮后, 停止捕获报文。
5. 点击“返回 (Clear)”按钮能够清空当前下方的编辑框控件中的内容。

## 6 实验现象和结论的分析 (验证程序正确性)

为了验证程序正确性, 我们用测试主机 ping 本地路由器 IP: 192.168.1.1。同时在 Wireshark 和我们的程序中对相应的网络接口卡进行监听捕获。测试我们的程序收到的结果是否与 Wireshark 中的一致。如果一致, 则程序运行正确; 若不一致, 则程序运行不正确。

图 2、3、4 证明了我们的程序的有效性。



```
DELL@DESKTOP-BPKFKRH ~  
λ ping 192.168.1.1  
  
正在 Ping 192.168.1.1 具有 32 字节的数据:  
来自 192.168.1.1 的回复: 字节=32 时间=13ms TTL=64  
来自 192.168.1.1 的回复: 字节=32 时间=3ms TTL=64  
来自 192.168.1.1 的回复: 字节=32 时间=1ms TTL=64  
来自 192.168.1.1 的回复: 字节=32 时间=7ms TTL=64  
  
192.168.1.1 的 Ping 统计信息:  
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),  
    往返行程的估计时间(以毫秒为单位):  
        最短 = 1ms, 最长 = 13ms, 平均 = 6ms
```

图 2: ping 192.168.1.1

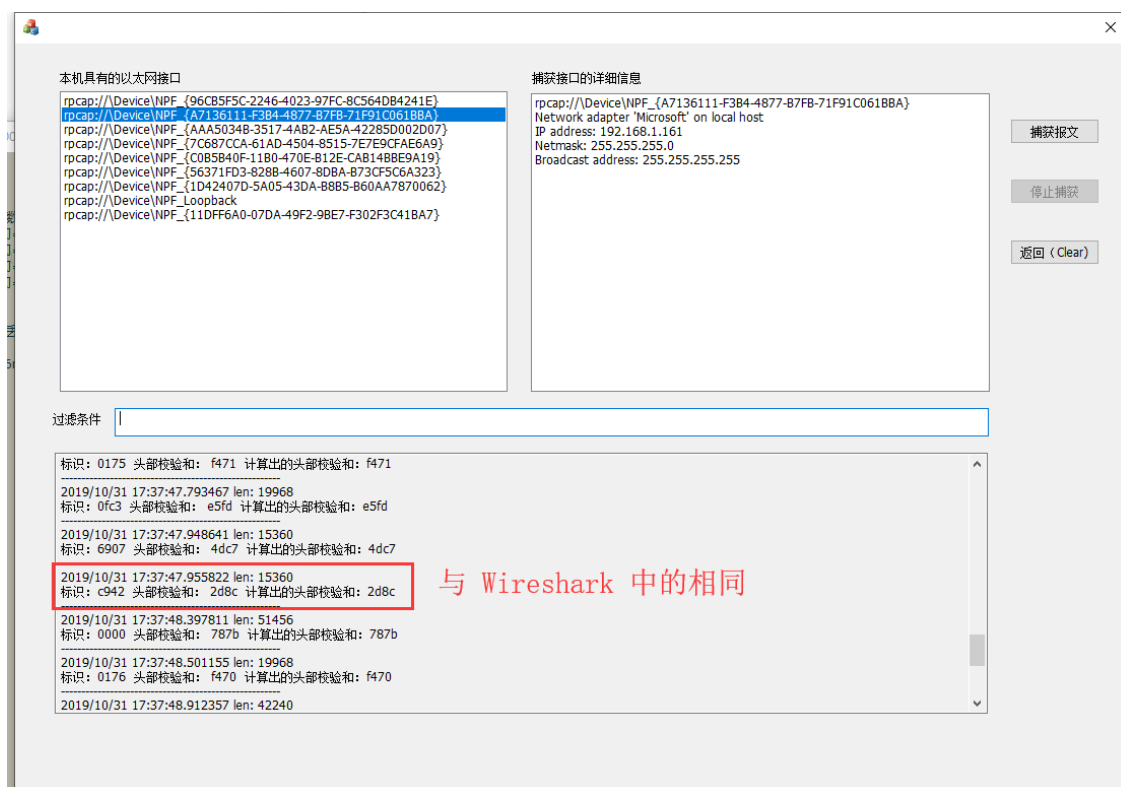


图 3: 我们的程序的显示

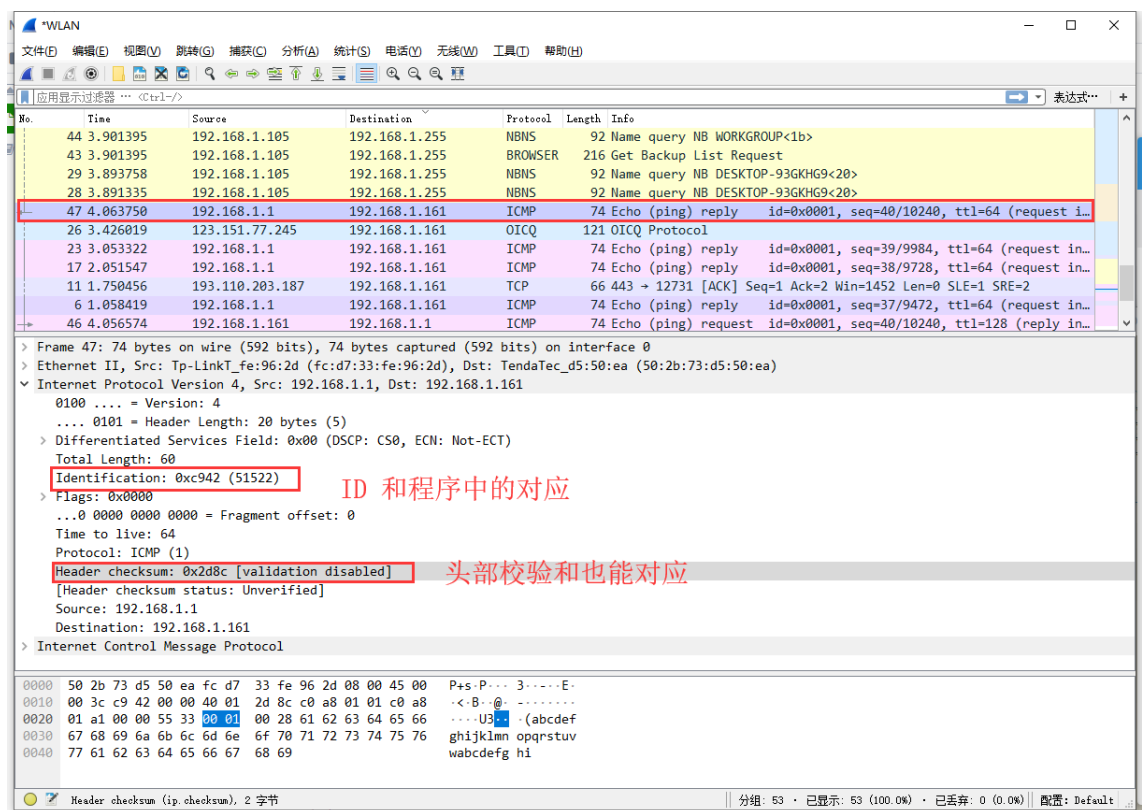


图 4: Wireshark 的显示

## A IPv4 header checksum

The **IPv4 header checksum** is a checksum used in version 4 of the Internet Protocol (IPv4) to detect corruption in the header of IPv4 packets. It is carried in the IP packet header, and represents the 16-bit result of summation of the header words.[1]

### A.1 Computation

The checksum calculation is defined in **RFC 791**

The checksum field is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

If there is no corruption, the result of summing the entire IP header, including checksum, should be zero. At each hop, the checksum is verified. Packets with checksum mismatch are discarded. The router must adjust the checksum if it changes the IP header (such as when decrementing the TTL).

### A.2 Examples

#### A.2.1 Calculating the IPv4 header checksum

Take the following truncated excerpt of an IPv4 packet. The header is shown in bold and the checksum is underlined.

**4500 0073 0000 4000 4011 b861 c0a8 0001**

**c0a8 00c7** 0035 e97c 005f 279f 1e4b 8180

For one's complement addition, each time a carry occurs, we must add a 1 to the sum. A carry check and correction can be performed with each addition or as a post-process after all additions. If another carry is generated by the correction, another 1 is added to the sum.

To calculate the checksum, we can first calculate the sum of each 16 bit value within the header, skipping only the checksum field itself. Note that these values are in hexadecimal notation.

$4500 + 0073 + 0000 + 4000 + 4011 + c0a8 + 0001 + c0a8 + 00c7 = 2479C$

The first digit is the carry count and is added to the sum:

$2 + 479C = 479E$  (if another carry is generated by this addition, another 1 must be added to the sum)

To obtain the checksum we take the one's complement of this result: B861 (as shown underlined in the original IP packet header).

### A.2.2 Verifying the IPv4 header checksum

When verifying a checksum, the same procedure is used as above, except that the original header checksum is not omitted.

$4500 + 0073 + 0000 + 4000 + 4011 + \underline{b861} + c0a8 + 0001 + c0a8 + 00c7 = 2fffd$

Add the carry bits:

$fffd + 2 = ffff$

Taking the ones' complement (flipping every bit) yields 0000, which indicates that no error is detected. IP header checksum does not check for the correct order of 16 bit values within the header.

## References

- [1] *IPv4 header checksum*. Sept. 2019. URL: [https://en.wikipedia.org/wiki/IPv4\\_header\\_checksum#cite\\_ref-2](https://en.wikipedia.org/wiki/IPv4_header_checksum#cite_ref-2).
- [2] 张建忠、徐敬东. 计算机网络技术与应用. 北京清华大学学研大厦 A 座: 清华大学出版社, 2019.