華中農業大學
HUAZHONG AGRICULTURAL UNIVERSITY

综合实训指导书

题目 网络流量在线分析系统的设计与实现

版本

院系 信息学院

编写时间

编写教师

综合实训指导书

一、实训目的

设计并实现一个网络流量的分析系统。该系统具有以下功能：（1）实时抓取网络数据。（2）网络协议分析与显示。（3）将网络数据包聚合成数据流，以源 IP、目的 IP、源端口、目的端口及协议等五元组的形式存储。（4）计算并显示固定时间间隔内网络连接（双向流）的统计量（如上行与下行的数据包数目，上行与下行的数据量大小等）。在这些统计数据的基础上分析不同网络应用的流量特征。

二、实训内容（实训的内容简介）

实训主要包括：

（1）能够实时抓取网络中的数据包。并实时显示在程序界面上。用户可自定义过滤条件以抓取所需要的数据包。

（2）分析各个网络协议格式，能够显示各协议字段的实际意义。例如，能够通过该程序反映 TCP 三次握手的实现过程。

（3）采用 Hash 链表的形式将网络数据以连接（双向流）的形式存储。

（4）计算并显示固定时间间隔内网络连接（双向流）的统计量（如上行与下行的数据包数目，上行与下行的数据量大小等）。例如，抓取一段时间（如 30 分钟）的网络流量，将该段时间以固定时长（如 1 分钟）为单位分成若干个时间片，计算网络连接在每一个时间片内的相关统计量。并在上述统计数据的基础上分析不同应用如 WEB、DNS、在线视频等服务的流量特征。注意，可根据实际的流量分析需要自己定义相关的统计量。

三、主要仪器及试材

硬件设备：

（1）台式计算机或笔记本计算机(含网络适配器)

软件设备：

（2）Windows 操作系统或 Linux 操作系统

（3）网络数据包捕获函数包，Windows 平台为 winpcap，Linux 平台下为 libpcap。这两个函数包可在网上下载安装。

（4）编程语言选用 C/C++。

四、实验方法与步骤（主体部分，学生按照此处的说明可以一步步完成整个实训）

4.1 实验环境的配置

（1）lipcap 或 winpcap 的下载和安装

Lipcap 下载地址为：http://www.tcpdump.org/release/libpcap-1.7.3.tar.gz

winpcap 下载地址为：http://www.winpcap.org/install/bin/WinPcap_4_1_3.exe

libpcap（Packet Capture library）即数据包捕获函数库。该库提供的 C 函数接口可用于需要捕获经过网络接口（只要经过该接口，目标地址不一定为本机）数据包的系统开发上。由 Berkeley 大学 Lawrence Berkeley National Laboratory 研究院的 Van Jacobson、Craig Leres 和 Steven McCanne 编写。该函数库支持 Linux、Solaris 和*BSD 系统平台。libpcap 主要由两部份组成：网络分接头(Network Tap)和数据过滤器(Packet Filter)。

Winpcap 为 lipcap 的 windows 版本，其功能与 libpcap 基本保持一致。

lipcap 的安装过程：

1）tar zxvf libpcap-1.7.3.tar.gz 解压文件，并将其放入自定义的安装目录。

2）打开网址：flex.sourceforge.net/ 下载 flex-2.5.35.tar.gz (1.40MB) 软件包，通过 tar zxvf flex-2.5.35.tar.gz 解压文件，并将其放入上述自定义的安装目录中。

注：如果没有编译安装此文件，在编译安装 libpcap 时，可能会出现“configure: error: Your operating system's lex is insufficient to compile libpcap.”的错误提示。

3）打开网址：ftp.gnu.org/gnu/bison/ 下载 bison-2.4.1.tar.gz (1.9MB) 软件包，通过 tar zxvf bison-2.4.1.tar.gz 解压文件，并将其放入上述自定义的安装目录中。

如果没有编译安装此文件，在编译安装 libpcap 时，可能会出现 "configure: WARNING: don't have both flex and bison; reverting to lex/yacc checking for capable lex... insufficient" 的错误提示。

4）打开网址：ftp.gnu.org/gnu/m4/ 下载 m4-1.4.13.tar.gz (1.2MB)软件包，通过 tar zxvf m4-1.4.13.tar.gz 解压文件，并将其放入上述自定义的安装目录中。

注：如果没有编译安装此文件，在编译安装 bison-2.4.1 时，就会出现 “configure: error: GNU M4 1.4 is required”的错误提示。

5）依次进入目录 m4-1.4.13，bison-2.4.1，flex-2.5.35，libpcap-1.0.0 并执行以下命令：

```
./configure
make
make instal
```

Winpcap 的安装过程：双击安装文件，根据提示安装即可。

（2）系统流程图

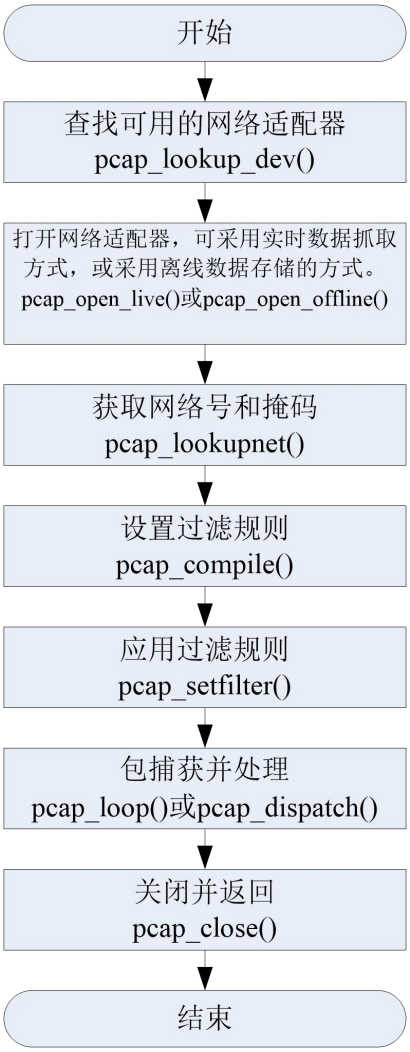

图 1 系统整体流程图

图 1 显示了系统的整体流程图。特别的，数据包捕获与处理部分是重点实现的内容，需实现网络流量分析的主要功能。注意，在系统实现的过程中根据需要采用多线程编程技术。

（3）流量分析功能实现。

1）首先，对抓取的每个数据包的各层的首部进行解析,并将解析结果进行显示。要求至少要解析以太网帧、IP 协议、TCP 协议和 UDP 协议。

a）数据链路层

抓到的数据包的以太网帧结构如下，头部长度为 14 字节：

Destination Address 目的 MAC 地址 [6 bytes]	Source Address 源 MAC 地址 [6 bytes]	Ethertype 以太网帧类型 [2 bytes]	Data 数据部分

图 2 以太网帧结构

MAC(Medium/Media Access Control)地址，用来表示互联网上每一个站点的标识符，采用

十六进制数表示，共 6 个字节（48 位）。每一张网卡都有唯一的 MAC 地址，因此以太网帧的目的 MAC 地址和源 MAC 地址。

以太网帧类型，该字段用来指明应用于帧数据字段的协议。下面列出了 EtherType 字段中常见的值及其对应的协议：

字段值（十六进制）	对应协议
0x0800	网际协议（IPv4）
0x0806	地址解析协议（ARP）
0x86DD	网际协议（IPv6）
0x880B	点对点协议（PPP）
0xffff	保留

图 3 常见 EtherType 值和对应协议

还需要特别注意的一点是，如果某一帧的长度小于 64 个字节，网卡会自动在该帧后添加填充位，使帧的长度大于 64 个字节。

b）网络层

IP 报文的结构如下：

<table>
<tr><td colspan="5">IP 报文格式</td></tr>
<tr><td>版本号
[4 bits]</td><td>首部长度
[4 bits]</td><td colspan="2">服务类型
[8 bits]</td><td>报文总长度
[16 bits]</td></tr>
<tr><td colspan="3">标识 [16 bits]</td><td>标志
[3 bits]</td><td>片偏移
[13 bits]</td></tr>
<tr><td colspan="2">生存时间 TTL
[8 bits]</td><td>协议类型 [8 bits]</td><td colspan="2">首部校验和 [16 bits]</td></tr>
<tr><td colspan="5">源地址 [32 bits]</td></tr>
<tr><td colspan="5">目的地址 [32 bits]</td></tr>
<tr><td colspan="5">选项（如果有）</td></tr>
<tr><td colspan="5">报文数据</td></tr>
</table>

图 4 IP 报文结构

学生在对 IP 报文进行解析时需注意：

首部长度，该字段指示当前 IP 首部的长度，所表示的数的单位是 32 bits（4 字节）。因此，由于该字段占 4 bits，最大值为 15，所以 IP 首部的长度最长为 60 个字节。但是，之后的报文总长度还是以字节为单位；

大多数情况下，IP 首部的长度是 20 字节，并且最少 20 个字节。但是一些情况下还会有**“选项”字段**，该字段的**长度是不确定**的。因此在分析数据包时应该考虑到。

c）传输层

传输层主要分析两种协议：TCP 与 UDP。

TCP(Transmission Control Protocol，传输控制协议)的数据结构如图 5 所示。

<table>
<tr><td colspan="3">源端口[16 bits]</td><td>目的端口[16 bits]</td></tr>
<tr><td colspan="4">序列号[32 bits]</td></tr>
<tr><td colspan="4">确认号[32 bits]</td></tr>
<tr><td>报头长度
[4 bits]</td><td>保留
[6 bits]</td><td>标志
[6 bits]</td><td>窗口[16 bits]</td></tr>
<tr><td colspan="3">校验和[16 bits]</td><td>紧急[16 bits]</td></tr>
<tr><td colspan="4">选项[可选]</td></tr>
<tr><td colspan="4">数据</td></tr>
</table>

图 5 TCP 报文结构

源端口：指定了发送端的端口。

目的端口：指定了接受端的端口号。

序列号：指明了段在即将传输的段序列中的位置。

确认号：成功收到段的序列号，确认序号包含发送确认的一端所期望收到的下一个序号。

TCP 长度：指定了段头的长度，取决于段头选项字段中设置的选项。

保留：指定了一个保留字段，以备将来使用。

标志：SYN（表示同步）、ACK（表示确认）、PSH（表示尽快地将数据送往接收进程）、RST（表示复位连接）、URG（表示紧急指针）、FIN（表示发送方完成数据发送）。

窗口：指定关于发送端能传输的下一段大小的指令。

校验和：校验和包含 TCP 段头和数据部分，用来校验段头和数据部分的可靠性。

紧急：指明段中包含紧急信息，只有当 URG 标志置 1 时紧急指针才有效。

选项：指定了公认的段大小，时间戳，选项字段的末端，以及指定了选项字段的边界选项。

UDP 的报文结构如图 6 所示。

<table>
<tr><td>源端口[16bits]</td><td>目的端口[16bits]</td></tr>
<tr><td>用户数据包的长度[16bits]</td><td>校验和[16bits]</td></tr>
<tr><td colspan="2">数据</td></tr>
</table>

图 6 UDP 用户数据报结构

源、目的端口：作用与 TCP 数据段中的端口号字段相同，用来标识源端和目标端的应用进程。

用户数据包的长度：标明 UDP 头部和 UDP 数据的总长度字节。

校验和：用来对 UDP 头部和 UDP 数据进行校验。注意，在校验和的计算中包括了 UDP 的伪首部。

2）**将捕获的数据包在进行解析的同时，将其存储在硬盘中为下一步的数据流的分析做准备。**

离线存储的 pcap 数据文件格式为：

<table>
<tr><td colspan="4">开头 24 字节，数据包文件（pcap）文件信息</td></tr>
<tr><td colspan="2">数据包文件（pcap）文件信息</td><td colspan="2">16 字节数据包信息</td></tr>
<tr><td colspan="2">数据包信息</td><td colspan="2">14 字节以太网帧头</td></tr>
<tr><td>以太网帧头</td><td colspan="3">网络层协议报头，如 IP 报头</td></tr>
<tr><td colspan="3">网络层协议报头</td><td>上层报头</td></tr>
<tr><td>报头</td><td colspan="3">数据域</td></tr>
<tr><td>数据域</td><td colspan="2">下一个 16 字节数据包信息</td><td>...</td></tr>
</table>

图 7 pcap 文件格式

文件开头是 24 字节的文件信息，开头是“d4c3 b2a1”，标识为 PCAP 文件，之后的 20 字节包含文件相关信息。紧接着就是各个包的具体数据。每个包开头是 16 字节的包信息，定义为 pcap_pkthdr 结构体，包含当前数据包抓取的时间和大小。接着是 14 个字节的以太网帧头。帧头后面是网络层、传输层等上层的协议报文，长度不定。可根据各报文头部来确定。一个包结束后，紧接着下一个包开始。

a）**读取离线存储的网络数据文件(pcap 文件)**。（通常抓取 30-60 分钟的网络流量）

系统涉及到保存数据到文件，以及从文件中读取数据。

在 C 语言中，安排一个指针变量指向一个文件，这个指针称为文件指针。通过文件指针就可对它所指的文件进行各种操作。定义说明文件指针的一般形式为：

FILE *指针变量标识符；

其中 FILE 是由系统定义的一个结构体，该结构中含有文件名、文件状态和文件当前位置等信息。例如：

FILE *fp；

表示 fp 是指向 FILE 结构的指针变量，通过 fp 即可找存放某个文件信息的结构变量，然后按结构变量提供的信息找到该文件，实施对文件的操作。

文件在进行读写操作之前要先打开，使用完毕后要关闭。打开文件，是指建立起指定文件的各种有关信息的结构，并使文件指针指向该文件，以便进行其它操作。关闭文件则断开指针与文件之间的联系，也就无法再对该文件进行操作。C 语言常用的文件操作函数见附录 1。

b）**基于 hash 表将网络数据以连接（双向流）的形式存储。**其实现流程如图 9 所示。**要求采用 Hash 链表的方式**保存一个时间段内的连接，有较高的查找性能。

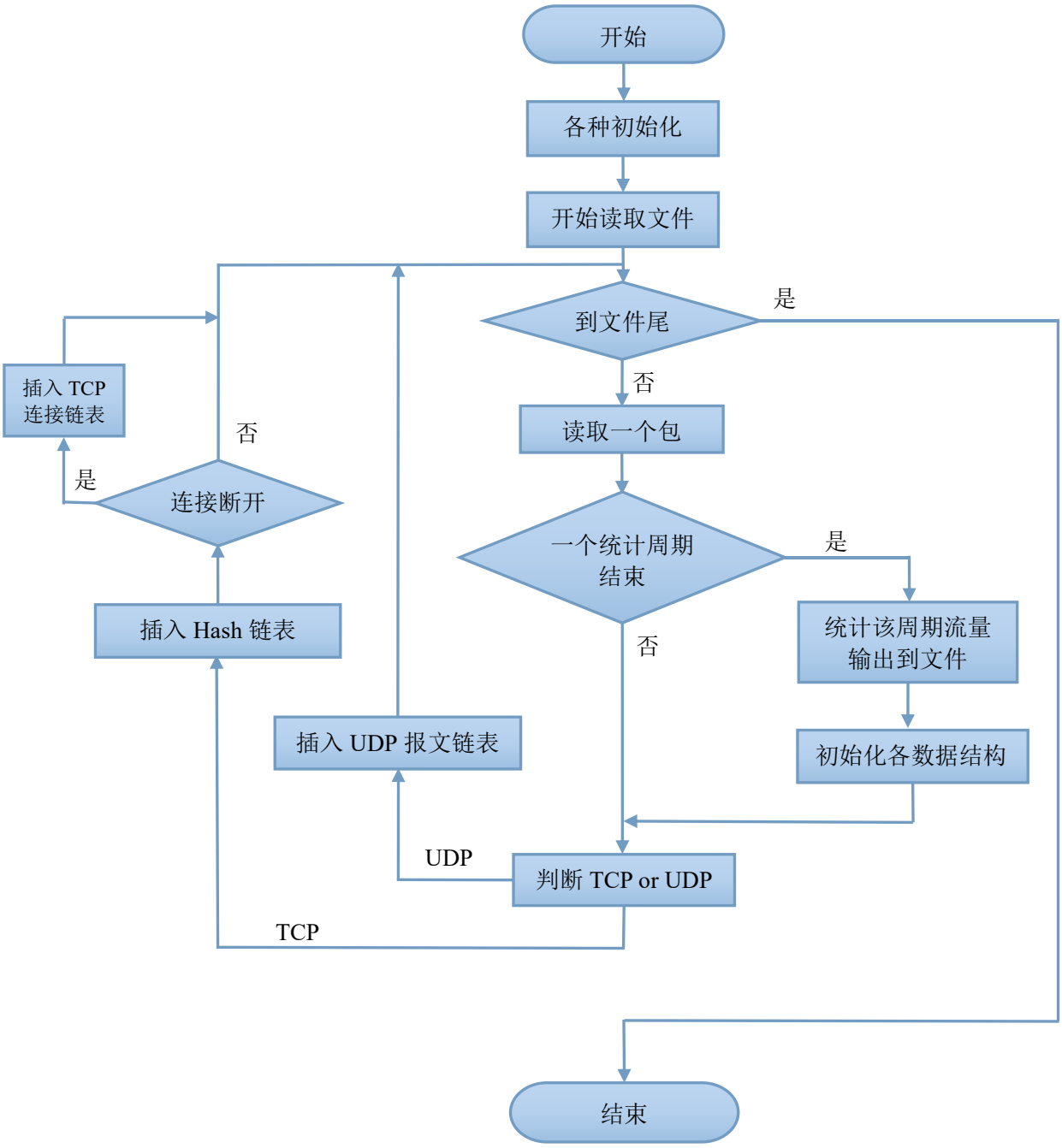

图 9　基于 hash 表的网络连接建立过程

c）**对 hash 链表进行分析并输出统计结果。**

计算并显示固定时间间隔内网络连接（双向流）的统计量（如上行与下行的数据包数目，上行与下行的数据量大小等）。例如，抓取一段时间（如 30 分钟）的网络流量，将该段时间以固定时长（如 1 分钟）为单位分成若干个时间片，计算网络连接在每一个时间片内的相关统计量。并在上述统计数据的基础上分析不同应用如 WEB、DNS、在线视频等服务的流量特

征（如上行与下行的数据包数目，上行与下行的数据量大小等，并可根据端口号判断应用类型）。注意，学生可根据实际的流量分析需要自己定义相关的统计量。

五、注意事项

1. 字节顺序

字节顺序是指占内存多于一个字节类型的数据在内存中的存放顺序，通常有小端、大端两种字节顺序。小端字节序指低字节数据存放在内存低地址处，高字节数据存放在内存高地址处；大端字节序是高字节数据存放在低地址处，低字节数据存放在高地址处。

网络字节序是 TCP/IP 规定好的一种数据表示格式，它与具体的 CPU 类型、操作系统无关，从而可以保证数据在不同主机之间传输时能被正确解释。网络字节顺序采用 big endian 排序方式。

在网络编程时，并不是什么时候都要考虑字节序问题。那么什么时候需要考虑呢？Intel CPU 使用的都是 little endian。实际上如果是应用层的数据，即对 TCP/IP 来说是透明的数据，不用考虑字节序的问题。因为接收端收到的顺序是和发送端一致的。但对于 TCP/IP 关心的数据（IP 地址、端口）来说就不一样了。例如指定一个端口号：

unsigned short port = 0x0012 （十进制 18）

把这个端口号传个 TCP/IP 建立一个 socket 连接。

```
sockaddr_in addr;
addr.sin_family = AF_INET; //使用互联网际协议，即 IP 协议
addr.sin_port = port;
addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
```

因为网络字节序是 big endian，即低地址存放的是数值的高位，所以 TCP/IP 实际上把这个 port 解释为 0x1200（十进制 4608）。本来打算是要在端口 18 建立连接的，但 TCP/IP 协议栈却在端口 4608 建立了连接。

当一个数据是应用层和 TCP/IP 都需要关心的时候，需要考虑这个数据的字节序。

字节序转换函数：

htons 把 unsigned short 类型从主机序转换到网络序

htonl 把 unsigned long 类型从主机序转换到网络序

ntohs 把 unsigned short 类型从网络序转换到主机序

ntohl 把 unsigned long 类型从网络序转换到主机序

htons 中 hton 代表 host to network， s 代表 unsigned short

char FAR * inet_ntoa(struct in_addr in); 将一个 IP 转换成一个互联网标准点分格式的字符串。

in_addr_t inet_addr(const char *cp); 将一个点分十进制的 IP 转换成一个长整数型数（u_long 类型）。返回值已是网络字节顺序，可以直接作为 internet 地址

2.基于 hash 链表的网络连接的存储

数据连接在本实验中定义为双向流。因此，在对 hash 链表进行比对时注意在正向查找不到时进行反向查找，即：将（源 IP，源 Port，目的 IP，目的 Port）变换为（目的 IP，目的 Port，源 IP，源 Port）后计算 hash 值，然后再 hash 链表中查找。

计算机在网络通信中，很大一部分是 TCP 连接。每抓到一个 TCP 报文，需要查找是否已经有该连接，如果有，则合并；若没有，则新加连接节点。

又考虑到网络通信中，数据量非常大，特别是在服务器类型的主机上。如果选择保存连接的数据结构的查找性能不高，则会造成程序低效、消耗过大。因此需要选择 Hash 链表保存一个时间段内的连接，有较高的查找性能。注意，在程序实现的过程中需要使用两个单链表——TCP 连接链表和 UDP 报文链表，分别保存一个时间段内已经结束的 TCP 连接，和抓取的 UDP 报文。

六、附录

附录 1 C 语言常用的文件操作函数

C 语言中，已经自带了文件操作相关的库函数，下面对需要用到的函数进行介绍：

• fopen(文件名,使用文件方式);

该函数用于打开文件。第一个参数即要打开的文件的“名称”，正确来说，应该是路径，相对于程序所在目录的相对路径，或者绝对路径。

第二个参数用于指定文件打开方式，见下表

• "r"（只读）

• "w"（只写）

• "a"（追加）

• "b"（读取二进制文件）

• "+"（可读可写）

以上各种方式可以相互组合。其中，“w”只写方式在打开文件时会把文件原有内容清空。因此，我们可以利用该模式清空某个文件。

• fclose(文件指针)

与 fread()相对应，用于关闭文件。编写程序时应及时关闭文件。

• 读写文件函数

• 字符读写函数 ：fgetc 和 fputc

• 字符串读写函数：fgets 和 fputs

• 数据块读写函数：fread 和 fwrite

• 格式化读写函数：fscanf 和 fprinf

对文件进行读写的过程中，会有一个文件位置指针，来指示当前所在文件位置距离文件头的位移量。注意，文件位置指针和文件指针是不一样的。

在上述的读写函数中，每读/写一次，文件位置指针会自动移动相应的位移。这点在按照格式读写文件时需要特别注意。

• fseek(文件指针，移动距离，参照位置)

该函数专门用来移动位置指针，利于按照约定格式读取数据。

要注意的是“参照位置”只能使用 3 个值：

SEEK_SET 文件开头　　0

SEEK_CUR位置指针当前位置　1

SEEK_END文件结尾　　2

• feof(文件指针);

该函数判断文件是否处于文件结束位置，如文件结束，则返回值为 1，否则为 0。

C 语言的文件操作库函数非常丰富，其余函数可自行查找。调用这些函数需要的头文件为 stdio.h。

附录 2 Libpcap 常用结构体、常量及常用的函数。

```
// pcap_next()方法执行后，pcap_pkthdr 类型的指针指向抓包的信息
struct pcap_pkthdr {
  struct timeval ts;       /* time stamp 时间 */
  bpf_u_int32 caplen;  /* length of portion present 包的数据长度 */
  bpf_u_int32 len;     /* length this packet 包的实际长度  */
};
// bpf_u_int32 即 unsigned int 类型

//timeval 结构
struct timeval{
  long tv_sec;          /* seconds 1900 之后的秒数 */
  long tv_usec;       /* and microseconds */
};
```

char *pcap_lookupdev(char *errbuf)

用于返回可被 pcap_open_live()或 pcap_lookupnet()函数调用的网络设备名指针。如果函数出错，则返回 NULL，同时 errbuf 中存放相关的错误消息。

int pcap_lookupnet(char *device, bpf_u_int32 *netp,bpf_u_int32 *maskp, char *errbuf)

用于获得指定网络设备的网络号和掩码。netp 参数和 maskp 参数都是 bpf_u_int32 指针。如果函数出错，则返回-1，同时 errbuf 中存放相关的错误消息。

pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)

u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)

返回指向下一个数据包的 u_char 指针。

int pcap_loop(pcap_t *p, int cnt,pcap_handler callback, u_char *user)

捕获并处理数据包，并调用指定的回调函数。

回调函数为 void 类型，函数名可任意指定。三个参数为：

u_char* argument，与 pcap_loop()方法的第三个参数对应，可用于传参；

const struct pcap_pkthdr* pkthdr，指向的内存保存了抓到的包的信息；

const u_char* packet_content，指向具体包数据

void pcap_close(pcap_t *p)

关闭 p 参数相应的文件，并释放资源。

int pcap_compile(pcap_t *p, struct bpf_program *fp,char *str, int optimize, bpf_u_int32 netmask)

将 str 参数指定的字符串编译到过滤程序中。

int pcap_setfilter(pcap_t *p, struct bpf_program *fp)

指定一个过滤程序。fp 参数是 bpf_program 结构指针，通常取自 pcap_compile()函数调用。出错时返回-1；成功时返回 0。

pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname)

打开用于保存捕获数据包的文件，用于写入。

void pcap_dump(u_char *user, struct pcap_pkthdr *h,u_char *sp)

向调用 pcap_dump_open()函数打开的文件输出一个数据包

void pcap_dump_close(pcap_dumper_t *p)

关闭相应的被打开文件。

附录 3 网络协议结构体定义

1）以太网首部

```
#define ETHER_LEN           14
#define ETHER_ADDR_LEN      6
#define ETHER_TYPE_LEN      2

typedef struct _ether_header{
    u_char host_dest[ETHER_ADDR_LEN];
```

```
    u_char host_src[ETHER_ADDR_LEN];
    u_short type;
#define ETHER_TYPE_MIN      0x0600
#define ETHER_TYPE_IP       0x0800
#define ETHER_TYPE_ARP      0x0806
#define ETHER_TYPE_8021Q    0x8100
#define ETHER_TYPE_BRCM     0x886c
#define ETHER_TYPE_802_1X   0x888e
#define ETHER_TYPE_802_1X_PREAUTH 0x88c7

}ether_header;
```

2）IP 首部

```
/* IPv4 header */
#define IP_LEN_MIN 20
typedef struct _ip_header{
    u_char  ver_ihl;      // Version (4 bits) + header length (4 bits)
    u_char  tos;          // Type of service
    u_short tlen;         // Total length
    u_short ident;        // Identification
    u_short flags_fo;     // Flags (3 bits) + Fragment offset (13 bits)
    u_char  ttl;          // Time to live
    u_char  proto;        // Protocol
#define IP_ICMP     1
#define IP_IGMP     2
#define IP_TCP      6
#define IP_UDP      17
#define IP_IGRP     88
#define IP_OSPF     89
    u_short crc;            // Header checksum
    u_int   saddr;          // Source address
    u_int   daddr;          // Destination address
}ip_header;
```

3）TCP 协议首部

```
/* TCP header */
#define TCP_LEN_MIN 20
typedef struct _tcp_header
{
    u_short th_sport;        // source port
    u_short th_dport;        // destination port
    u_int   th_seq;          // sequence number field
    u_int   th_ack;          // acknowledgement number field
    u_char  th_len:4;        // header length
```

```
    u_char  th_x2:4;         // unused
    u_char  th_flags;
  #define TH_FIN 0x01
  #define TH_SYN 0x02
  #define TH_RST 0x04
  #define TH_PSH 0x08
  #define TH_ACK 0x10
  #define TH_URG 0x20
    u_short th_win;           /* window */
    u_short th_sum;           /* checksum */
    u_short th_urp;           /* urgent pointer */
  }tcp_header;
```

4）UDP 协议

```
  /* UDP header */
  #define UDP_LEN 8

  typedef struct _udp_header{
    u_short uh_sport;          // Source port
    u_short uh_dport;          // Destination port
    u_short uh_len;            // Datagram length
    u_short uh_sum;            // Checksum
  }udp_header;
```

附录四 参考网络资源

1）Libpcap 官网：http://www.tcpdump.org/

2）Winpcap 官网：http://www.winpcap.org/

3）字节顺序_百度百科

http://baike.baidu.com/link?url=bQPKqJkMkczUieXcIGmqtbklhM7lY5cVZsBI12hycjo0IntItSAR3fdP-5mkbAR1

4）libpcap 详解-chinaltang-ChinaUnix 博客

http://blog.chinaunix.net/uid-21556133-id-120228.html

七、参考代码：

/ protocol.h**

structs of ethernet, ip, tcp, udp

*/

#define PCAP_HEADER_LEN 24

```
#define PACKET_HEADER_LEN 16

/* ============= Ethernet ============= */
#define ETHER_LEN                  14
#define ETHER_ADDR_LEN          6
#define ETHER_TYPE_LEN          2

#define ETHER_DEST_OFFSET     (0 * ETHER_ADDR_LEN)
#define ETHER_SRC_OFFSET       (1 * ETHER_ADDR_LEN)
#define ETHER_TYPE_OFFSET     (2 * ETHER_ADDR_LEN)

typedef struct _ether_header{
      u_char host_dest[ETHER_ADDR_LEN];
      u_char host_src[ETHER_ADDR_LEN];
      u_short type;
#define ETHER_TYPE_MIN          0x0600
#define ETHER_TYPE_IP            0x0800
#define ETHER_TYPE_ARP          0x0806
#define ETHER_TYPE_8021Q       0x8100
#define ETHER_TYPE_BRCM         0x886c
#define ETHER_TYPE_802_1X      0x888e
#define ETHER_TYPE_802_1X_PREAUTH 0x88c7
}ether_header;



/*============== IP ================*/
#define IP_LEN_MIN 20

/* IPv4 header */
typedef struct _ip_header{
      u_char    ver_ihl;              // Version (4 bits) + Internet header length (4 bits)
```

```
    u_char    tos;                // Type of service
    u_short tlen;                 // Total length
    u_short ident;                // Identification
    u_short flags_fo;             // Flags (3 bits) + Fragment offset (13 bits)
    u_char    ttl;                // Time to live
    u_char    proto;              // Protocol
    #define IP_ICMP        1
    #define IP_IGMP        2
    #define IP_TCP         6
    #define IP_UDP         17
    #define IP_IGRP        88
    #define IP_OSPF        89
    u_short crc;                  // Header checksum
    u_int     saddr;              // Source address
    u_int     daddr;              // Destination address
}ip_header;


/*================ TCP ================*/
#define TCP_LEN_MIN 20
typedef struct _tcp_header
{
    u_short th_sport;             // source port
    u_short th_dport;             // destination port
    u_int     th_seq;             // sequence number field
    u_int     th_ack;             // acknowledgement number field
    u_char    th_len:4;           // header length
    u_char    th_x2:4;            // unused
    u_char    th_flags;
#define TH_FIN 0x01
#define TH_SYN     0x02
#define TH_RST     0x04
```

```
#define TH_PSH      0x08
#define TH_ACK      0x10
#define TH_URG      0x20
    u_short th_win;             /* window */
    u_short th_sum;             /* checksum */
    u_short th_urp;             /* urgent pointer */
}tcp_header;    //*/

/*================ UDP ==================*/
#define UDP_LEN 8
typedef struct _udp_header{
    u_short uh_sport;               // Source port
    u_short uh_dport;               // Destination port
    u_short uh_len;                 // Datagram length
    u_short uh_sum;                   // Checksum
}udp_header;
```

/catch_packet.c**

```
#include <stdio.h>
#include <pcap.h>
#include <pthread.h>
//#include "protocol.h"
```

```
typedef struct _argument
{
	pcap_t *handle;
	int timeLen;
}argument;

void *thread_clock(void *argv)
{
	pcap_t *handle = ((argument*)argv)->handle;
	int timeLen = ((argument*)argv)->timeLen;    // set time
	sleep(timeLen);
	pcap_breakloop(handle);
}

void cb_getPacket(u_char *dumpfile, const struct pcap_pkthdr *pkthdr, const u_char *packet)
{
	// ip_header *seg_ip = (ip_header*)(package + ETHER_LEN);
	pcap_dump(dumpfile, pkthdr, packet);

	static int id = 0;
	printf(".    ");
	if(++id % 30 == 0)
	{
		printf("\n");
	}
}

int main(int argc, char const *argv[])
{
	char			*dev, errbuf[PCAP_ERRBUF_SIZE];
	pcap_t			*dev_handle;
```

```
bpf_u_int32                net, mask;
char                        packet_filter[] = "ip";
struct bpf_program    fcode;

dev = pcap_lookupdev(errbuf);
if(dev == NULL){
        printf("No Device:%s\n", errbuf);
        return 0;
} // */
/*char *wlan_dev = "wlan0";
dev = wlan_dev; // */
printf("开始抓取数据，设备:%s\n", dev);

dev_handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
if(dev_handle == NULL){
        printf("pcap_open_live:%s\n", errbuf);
        return 0;
}
//args->handle = dev_handle;

pcap_lookupnet(dev, &net, &mask, errbuf);

//compile the filter
if (pcap_compile(dev_handle, &fcode, packet_filter, 1, mask) <0 )
{
        printf("\nUnable to compile the packet filter. Check the syntax.\n");
        return 0;
}
//set the filter
if (pcap_setfilter(dev_handle, &fcode) < 0)
{
        printf("\nError setting the filter.\n");
```

```
        return 0;
    }

    // open file to save pcap
    pcap_dumper_t *dumpfile;
    dumpfile = pcap_dump_open(dev_handle, "traffic.data");
    if(dumpfile == NULL){
        printf("\nError opening output file\n");
        return 0;
    }

    // build a new thread
    pthread_t ptClock;
    argument args;
    args.handle = dev_handle;
    int argv_time = atoi(argv[1]);
    args.timeLen = (argv_time > 0) ? argv_time : 60;
    printf("抓取时长：%d s\n", argv_time);
    if(pthread_create(&ptClock, NULL, thread_clock, &args))
    {
        printf("pthread_create(): Error!\n");
        return -1;
    }
    pcap_loop(dev_handle, -1, cb_getPacket, (u_char*)dumpfile);

    // close all handle
    pcap_dump_close(dumpfile);
    pcap_close(dev_handle);
    printf("\nDone!\n");
    return 0;
}
```

```
/** pcap_analysis.c*/
#include <stdio.h>
#include <malloc.h>
#include <time.h>
#include <string.h>
#include <pcap.h>
#include "protocol.h"


//timeval 结构
typedef struct _shh_timeval{
    int tv_sec;             /* seconds 1900 之后的秒数  */
    int tv_usec;           /* and microseconds */
}shh_timeval;

// pcap_next()方法执行后，pcap_pkthdr 类型的指针指向抓包的信息
typedef struct _shh_pkthdr {
    shh_timeval ts;    /* time stamp  时间  */
    bpf_u_int32 caplen; /* length of portion present  包的数据长度？？  */
    bpf_u_int32 len;       /* length this packet (off wire)  包的实际长度    */
}shh_pkthdr;

typedef struct _net5set
{
    u_int           sip;
    u_short         sport;
    u_int           dip;
    u_short         dport;
    u_char          protocol;
}net5set;

typedef struct _net_link_node
```

```
{
    net5set nln_5set;
    int         nln_upl_size;
    int         nln_downl_size;
    int         nln_upl_pkt;
    int         nln_downl_pkt;
    u_char    nln_status;
    #define CLOSED           0x00
    #define SYN_SENT       0x01       // client sent SYN
    #define SYN_RECVD     0x02       // recieve SYN, and send SYN ACK
    #define ESTABLISHED 0x03       // client get SYN & ACK, server get ACK

    #define FIN_WAIT_1    0x04       // client send FIN
    #define CLOSE_WAIT    0x05       // server recv FIN, and send ACK
    #define FIN_WAIT_2    0x06       // client recv ACK
    #define LAST_ACK       0x07       // server send FIN
    #define TIME_WAIT     0x08       // client recv FIN
    // CLOSED: client send ACK, server recv ACK
    #define UNDEFINED     0xff
    struct    _net_link_node *next;
}net_link_node, *p_net_link;

typedef struct _net_link_header
{
    int count_conn;
    int count_upl_pkt;
    int count_downl_pkt;
    int count_upl;
    int count_downl;
    p_net_link link;
}net_link_header;
```

```
#define IPTOSBUFFERS        12
static char *iptos(bpf_u_int32 in)
{
        static char output[IPTOSBUFFERS][3*4+3+1];
        static short which;
        u_char *p;

        p = (u_char *)&in;
        which = (which + 1 == IPTOSBUFFERS ? 0 : which + 1);
        sprintf(output[which], "%d.%d.%d.%d", p[0], p[1], p[2], p[3]);
        return output[which];
}

char *long2time(long ltime)
{
        time_t t;
        struct tm *p;
        static char s[100];

        t = ltime;
        p = gmtime(&t);

        strftime(s, sizeof(s), "%Y-%m-%d %H:%M:%S", p);
        return s;
}

// 需要三个链表，一个哈希链表，保存处于连接状态的包
// 另两个链表分别保存 tcp 和 udp 的流量
net_link_header *FLowLink_TCP;
net_link_header *FLowLink_UDP;
```

```
/* ========== hash table ============= */
#define HASH_TABLE_SIZE 0xffff
p_net_link HashTable[HASH_TABLE_SIZE];

void init_flowLink(net_link_header *head)
{
	head->count_conn		= 0;
	head->count_upl_pkt		= 0;
	head->count_downl_pkt	= 0;
	head->count_upl			= 0;
	head->count_downl		= 0;
	head->link				= NULL;
}

void add_to_flowLink(net_link_header *head, const net_link_node *theNode)
{
	net_link_node *newNode = (net_link_node *)malloc(sizeof(net_link_node));
	memcpy(newNode, theNode, sizeof(net_link_node));

	head->count_conn ++;
	head->count_upl_pkt		+= newNode->nln_upl_pkt;
	head->count_downl_pkt	+= newNode->nln_downl_pkt;
	head->count_upl			+= newNode->nln_upl_size;
	head->count_downl		+= newNode->nln_downl_size;

	newNode->next = head->link;
	head->link = newNode;
}

void clear_flowLink(net_link_header *head)
{
	if( head->link == NULL ){ return;}
```

```
        net_link_node *pTemp1 = NULL;
        net_link_node *pTemp2 = NULL;

        pTemp1 = head->link;
        pTemp2 = pTemp1->next;
        while( pTemp2 != NULL )
        {
            free(pTemp1);
            pTemp1 = pTemp2;
            pTemp2 = pTemp1->next;
        }
        free(pTemp1);

        head->link = NULL;
    }

    void parse_flowLink_TCP(FILE *fOutput)
    {
        fprintf(fOutput, "TCP 连接个数：\t%d\n", FLowLink_TCP->count_conn);
        fprintf(fOutput, "TCP 数据包个数： \t%d\n", FLowLink_TCP->count_upl_pkt +
FLowLink_TCP->count_upl_pkt);
        fprintf(fOutput, "TCP 数据总流量： \t%d bytes\n", FLowLink_TCP->count_upl +
FLowLink_TCP->count_downl);
        fprintf(fOutput, "TCP 数据上传量：\t%d bytes\n", FLowLink_TCP->count_upl);
        fprintf(fOutput, "TCP 数据下载量：\t%d bytes\n", FLowLink_TCP->count_downl);
        fprintf(fOutput, "----------------------\n");

        net_link_node *pTemp = NULL;
        pTemp = FLowLink_TCP->link;
        while( pTemp != NULL )
        {
```

```
            fprintf(fOutput, "%s\t%u\t", iptos(pTemp->nln_5set.sip), pTemp->nln_5set.sport);
            fprintf(fOutput, "==>\t%s\t%u\t", iptos(pTemp->nln_5set.dip),
pTemp->nln_5set.dport);
            fprintf(fOutput, "上传包数量：%d\t", pTemp->nln_upl_pkt);
            fprintf(fOutput, "下载包数量：%d\t", pTemp->nln_downl_pkt);
            fprintf(fOutput, "upload：%d bytes\t", pTemp->nln_upl_size);
            fprintf(fOutput, "download：%d bytes\t", pTemp->nln_downl_size);
            fprintf(fOutput, "\n");
            pTemp = pTemp->next;
        }

        clear_flowLink(FLowLink_TCP);

    }

    void parse_flowLink_UDP(FILE *fOutput)
    {
        fprintf(fOutput, "UDP 数据包个数： \t%d\n", FLowLink_UDP->count_upl_pkt +
FLowLink_UDP->count_upl_pkt);
        fprintf(fOutput, "UDP 数据流量： \t%d bytes\n", FLowLink_UDP->count_upl +
FLowLink_UDP->count_downl);
        clear_flowLink(FLowLink_UDP);
    }



    u_short get_ushort_net(u_short virtu)
    {
        return (u_short)(virtu >> 8 | virtu << 8);
    }
```

```
u_short get_hash(const net5set *theSet)
{
	u_int srcIP = theSet->sip;
	u_int desIP = theSet->dip;
	u_int port    = (u_int)(theSet->sport * theSet->dport);
	u_int res      = (srcIP^desIP)^port;
	u_short hash= (u_short)((res & 0x00ff)^(res >> 16));
	return hash;
}

void add_to_hashTable(u_short hash, const net_link_node *theNode, u_char flags)
{
	net_link_node *HashNode = (net_link_node *)malloc(sizeof(net_link_node));
	memcpy(HashNode, theNode, sizeof(net_link_node));

	if(HashTable[hash] == NULL)
	{
		HashTable[hash] = HashNode;
		return;
	}
	net_link_node *pTemp = HashTable[hash];
	net_link_node *pBack = NULL;
	int isSame_up = 0;
	int isSame_down = 0;
	while(pTemp != NULL)
	{
		isSame_up = (pTemp->nln_5set.sip == HashNode->nln_5set.sip)
				&& (pTemp->nln_5set.dip == HashNode->nln_5set.dip)
				&& (pTemp->nln_5set.sport == HashNode->nln_5set.sport)
				&& (pTemp->nln_5set.dport == HashNode->nln_5set.dport);
```

```
isSame_down = (pTemp->nln_5set.dip == HashNode->nln_5set.sip)
        && (pTemp->nln_5set.sip == HashNode->nln_5set.dip)
        && (pTemp->nln_5set.dport == HashNode->nln_5set.sport)
        && (pTemp->nln_5set.sport == HashNode->nln_5set.dport);
if( isSame_up )
{
    pTemp->nln_upl_size += HashNode->nln_upl_size;
    pTemp->nln_upl_pkt ++;
    if(pTemp->nln_status == ESTABLISHED && (flags & TH_FIN) )
    {
        pTemp->nln_status = FIN_WAIT_1;
    }
    else if (pTemp->nln_status == TIME_WAIT && (flags & TH_ACK))
    {
        pTemp->nln_status = CLOSED;
        if(pBack == NULL)
        {
            HashTable[hash] = NULL;
        }
        else
        {
            pBack->next = pTemp->next;
        }
        add_to_flowLink(FLowLink_TCP, pTemp);
        free(pTemp);
    }
    else if(pTemp->nln_status == CLOSE_WAIT && (flags & TH_FIN))
    {
        pTemp->nln_status = LAST_ACK;
    }
    free(HashNode);
    break;
```

```
}
else if( isSame_down )
{
    pTemp->nln_downl_size += HashNode->nln_upl_size;
    pTemp->nln_downl_pkt ++;
    if(pTemp->nln_status == ESTABLISHED && (flags & TH_FIN))
    {
        pTemp->nln_status = CLOSE_WAIT;
    }
    else if(pTemp->nln_status == LAST_ACK && (flags & TH_ACK))
    {
        pTemp->nln_status = CLOSED;
        if(pBack == NULL)
        {
            HashTable[hash] = NULL;
        }
        else
        {
            pBack->next = pTemp->next;
        }
        add_to_flowLink(FLowLink_TCP, pTemp);
        free(pTemp);
    }
    else if(pTemp->nln_status == FIN_WAIT_1 && (flags & TH_ACK))
    {
        pTemp->nln_status = FIN_WAIT_2;
    }
    else if(pTemp->nln_status == FIN_WAIT_2 && (flags & TH_FIN))
    {
        pTemp->nln_status = TIME_WAIT;
    }
```

```
                free(HashNode);
                break;
            }
            pBack = pTemp;
            pTemp = pTemp->next;
        }
        if(pTemp == NULL)
        {
            pBack->next = HashNode;
        }
}

void clear_hashTable()
{
        int i = 0;
        net_link_node *pTemp1 = NULL;
        net_link_node *pTemp2 = NULL;
        for(i = 0; i < HASH_TABLE_SIZE; i++)
        {
            if(HashTable[i] == NULL){ continue;}

            pTemp1 = HashTable[i];
            while(pTemp1 != NULL)
            {
                pTemp2 = pTemp1->next;
                add_to_flowLink(FLowLink_TCP, pTemp1);
                free(pTemp1);
                pTemp1 = pTemp2;
            }
            HashTable[i] = NULL;
        }
}
```

```
/*
在以太网中，规定最小的数据包为 64 个字节，如果数据包不足 64 字节，则会由网卡填充。
*/

int main(int argc, char const *argv[])
{
    char *file_output = "result.data";
    FILE *fOutput = fopen(file_output, "w");
    fclose(fOutput);              // clear file
    fOutput = fopen(file_output, "a+");

    char *filename = "traffic.data";
    fprintf(fOutput, "数据文件：%s\n", filename);

    printf("载入文件...\n");
    FILE *fp = fopen(filename, "r");

    shh_pkthdr      *pkthdr     = (shh_pkthdr *)malloc(sizeof(shh_pkthdr));
    ether_header    *segEther   = (ether_header*)malloc(sizeof(ether_header));
    ip_header       *segIP      = (ip_header*)malloc(sizeof(ip_header));
    tcp_header      *segTCP     = (tcp_header*)malloc(sizeof(tcp_header));
    udp_header      *segUDP     = (udp_header*)malloc(sizeof(udp_header));
    net5set         *Cur5Set    = (net5set *)malloc(sizeof(net5set));
    net_link_node   *LinkNode   = (net_link_node *)malloc(sizeof(net_link_node));

    FLowLink_TCP = (net_link_header *)malloc(sizeof(net_link_header));
    init_flowLink(FLowLink_TCP);
    FLowLink_UDP = (net_link_header *)malloc(sizeof(net_link_header));
```

```
init_flowLink(FLowLink_UDP);

long       fileLen         = 0;
int        pktLen          = 0;       // pktLen = Ether + IP
int        trailerLen    = 0;
u_short ipLen_real    = 0;
u_short ipLen_total = 0;
u_short tcpLen_real = 0;
u_short dataLen         = 0;

// get length of file
fseek(fp, 0, SEEK_END);
fileLen = ftell(fp);
fseek(fp, PCAP_HEADER_LEN, SEEK_SET);
//  移动文件位置指针。
// If successful, the function returns zero.
// Otherwise, it returns non-zero value.
// SEEK_SET:文件开头;SEEK_CUR:当前位置;SEEK_END:文件结尾

fread(pkthdr, PACKET_HEADER_LEN, 1, fp);
fseek(fp, - PACKET_HEADER_LEN, SEEK_CUR);
int tstamp_start       = pkthdr->ts.tv_sec;
int tstamp_offset      = tstamp_start;
int tstamp_now          = tstamp_start;
int cycle                    = atoi(argv[1]);
cycle = (cycle > 0) ? cycle : 10;
fprintf(fOutput, "分析周期：%d s\n", cycle);

int i = 0;
while( ftell(fp) > 0 &&    ftell(fp) < fileLen )
{
      fread(pkthdr, PACKET_HEADER_LEN, 1, fp);
```

```
pktLen = pkthdr->caplen;
tstamp_now = pkthdr->ts.tv_sec;
if(tstamp_now - tstamp_offset >= cycle)
{
    fprintf(fOutput, "\n\n>>>>> 时间段：%s", long2time(tstamp_offset));
    fprintf(fOutput, " --> %s\n", long2time(tstamp_offset + cycle));

    fprintf(fOutput, "----------------------\n");
    clear_hashTable();
    parse_flowLink_UDP(fOutput);
    init_flowLink(FLowLink_UDP);

    fprintf(fOutput, "----------------------\n");
    parse_flowLink_TCP(fOutput);
    init_flowLink(FLowLink_TCP);
    tstamp_offset = tstamp_now;

}
//printf("%d\t", pktLen);
//printf("\n%d\t", ++i);

fread(segEther, ETHER_LEN, 1, fp);
if( get_ushort_net(segEther->type) != ETHER_TYPE_IP )
{
    //printf("------\t");
    fseek(fp, pktLen - ETHER_LEN, SEEK_CUR);
    continue;
}

fread(segIP, IP_LEN_MIN, 1, fp);
ipLen_real = (segIP->ver_ihl & 0x0f)*4;
ipLen_total = get_ushort_net(segIP->tlen);
```

```
trailerLen = pktLen - ETHER_LEN - ipLen_total;
fseek(fp, ipLen_real - IP_LEN_MIN, SEEK_CUR);

if( segIP->proto != IP_TCP && segIP->proto != IP_UDP )
{
    //printf("------\t");
    fseek(fp, ipLen_total - ipLen_real + trailerLen, SEEK_CUR);
    continue;
}

Cur5Set->sip = segIP->saddr;
Cur5Set->dip = segIP->daddr;
Cur5Set->protocol = segIP->proto;
//printf("src:%s\t", iptos(Cur5Set->sip));
//printf("des:%s\t", iptos(Cur5Set->dip));

if(segIP->proto == IP_TCP)
{
    //printf("TCP\t");
    fread(segTCP, TCP_LEN_MIN, 1, fp);
    tcpLen_real = (((segTCP->th_len)>>4) & 0x0f) * 4;
    dataLen = ipLen_total - ipLen_real - tcpLen_real;

    Cur5Set->sport = get_ushort_net(segTCP->th_sport);
    Cur5Set->dport = get_ushort_net(segTCP->th_dport);

    fseek(fp, (tcpLen_real - TCP_LEN_MIN) + dataLen + trailerLen,
SEEK_CUR);
}
else if(segIP->proto == IP_UDP)
{
    //printf("UDP\t");
```

```
            fread(segUDP, UDP_LEN, 1, fp);
            dataLen = ipLen_total - ipLen_real - UDP_LEN;

            Cur5Set->sport = get_ushort_net(segUDP->uh_sport);
            Cur5Set->dport = get_ushort_net(segUDP->uh_dport);

            fseek(fp, dataLen + trailerLen, SEEK_CUR);
        }
        LinkNode->nln_5set          = *Cur5Set;
        LinkNode->nln_upl_size    = dataLen;
        LinkNode->nln_downl_size= 0;
        LinkNode->nln_upl_pkt     = 1;
        LinkNode->nln_downl_pkt = 0;
        LinkNode->nln_status       = ESTABLISHED;
        LinkNode->next                = NULL;

        if(segIP->proto == IP_TCP)
        {
            add_to_hashTable(get_hash(Cur5Set), LinkNode, segTCP->th_flags);
        }
        else
        {
            add_to_flowLink(FLowLink_UDP, LinkNode);
        }
    }
    fprintf(fOutput, "\nover\n");

    free(pkthdr);
    free(segEther);
    free(segIP);
    free(segTCP);
    free(segUDP);
```

```
    free(Cur5Set);
    free(LinkNode);
    free(FLowLink_TCP);
    free(FLowLink_UDP);
    fclose(fOutput);

    printf("Done!\n");
    return 0;
}
```