Project 2: Network Forensics

Problem Description

Network forensics is "the monitoring and analysis of computer network traffic for the purposes of information gathering, legal evidence, or intrusion detection". In this project, you will build a network forensic system by following step-by-step instructions. At high level, the system takes as input a network packet trace captured over the wire, parses the packets (Task 1), assembles TCP flows (Task 2), and reconstructs HTTP conversations (Task 3). The final goal is to extract images downloaded from web servers (Task 4). Each of Tasks 2, 3, and 4 relies on the completion of the previous task, so you should work on the tasks in their original order.

Task 1: Capture and Parse PCAP trace

`tcpdump` is the *de-facto* tool for live packet capture. Its basic usage is as follows:

```
tcpdump -i eth0 -w ./test.pcap
```

The above command has two switches. The `-i` switch specifies which network interface to monitor. For example, on Linux-based systems, `eth0` is usually the main Ethernet interface, and `wlan0` is often the 802.11 (Wi-Fi) interface, *etc.* You can use the `ifconfig` command to list all interfaces. The `-w` switch specifies the name of the file which captured packets will be written to. By default, the entire packet (including all headers and the payload) are captured. Please try it yourself: open a Linux terminal, type the above command to start capture. Generate some network traffic (*e.g.,* browse some web pages) before pressing `Ctrl+C` to stop capture.

The captured file is in PCAP (Packet CAPture) format. It is binary so a PCAP file cannot be directly displayed using, for example, `cat`. Instead, you are strongly encouraged to use Wireshark (`https://www.wireshark.org/`), a powerful packet analyzer, to view the trace. Wireshark pro-vides numerous features such as protocol analysis, packet filtering, and packet capture. What you need to implement in this project is actually a subset of Wireshark's functionalities. Please familiar yourself with Wireshark, as it will be very helpful for debugging your code in this project.

PCAP has a very simple format. It begins with a fixed 24-byte header, followed by records of captured packets. Each record consists of a 16-byte header followed by the actual packet data.

The detailed format description can be found at `https://wiki.wireshark.org/Development/LibpcapFileFormat`. Some libraries (*e.g.,* `libpcap`) provide APIs for parsing PCAP files, but you need to implement this by yourself in this project.

Now let's discuss the objective of Task 1. Your program reads a PCAP file from `stdin` and computes some statistics. You output to `stdout` a single line consisting of five numbers:

1. The total number of packets in the trace. This includes *all* packets: IP and non-IP packets.
2. The total number of IP packets (IPv4 only, IPv6 packets will not appear in this project).
3. The total number of TCP packets.
4. The total number of UDP packets.
5. The total number of TCP connections. Recall a TCP connection is a four-tuple of {source IP, source port, destination IP, destination port}. In other words, there is a one-to-one mapping between a TCP connection and its four-tuple. Also recall that a TCP connection is bidirectional so (IP1, Port1)→(IP2, Port2) and (IP2, Port2)→(IP1, Port1) belong to the same connection.

For example, if the trace contains 1,000 packets, 900 IP packets, 800 TCP packets, 50 UDP packets, and 15 TCP connections, your output must be ("␣" denotes a white space):

`1000␣900␣800␣50␣15\n`

The five numbers are separated by one space. There is no trailing space after the last number. You can test your code by capturing some traces yourself, and then compare your results with the results reported by Wireshark.

Note that a TCP connection may be *partially* present in the trace. For example, if you start `tcpdump` after TCP handshake happens, apparently you will not see the handshake in the trace. When counting TCP connections, you must include partially captured connections.

Throughout this project, you may assume the input PCAP file is always valid, and each packet always has a 14-byte Ethernet header. You can also assume the input PCAP file is no larger than 30MB, and contains no more than 30,000 packets.

Task 2: Assemble TCP Connections

In this task, we perform TCP connection assembly *i.e.,* extracting TCP data streams from the packet trace. The basic idea is simple. Assume a connection has three packets in one direction:

Packet P_1: Sequence number = 100–102, Data = "ABC"
Packet P_2: Sequence number = 103–107, Data = "DEFGH"
Packet P_3: Sequence number = 108–109, Data = "IJ"

The assembled data stream is the concatenation of payload data in its sequence numbers' ascending order. In the above example, the data stream is "ABCDEFGHIJ", with the sequence numbers ranging from 100 to 109.

There are additional complexities though. Since TCP ensures reliable and in-order delivery, it needs to deal with situations such as packet retransmission, duplication, and out-of-order. We discuss three scenarios below.

First, packets can be out-of-order. In the above example, the order of the three packets appearing in the trace can be, for example, P_1, P_2, P_3, P_1, P_3, P_2, or P_3, P_2, P_1.

Second, there can be duplicate packets due to spurious retransmission. For example, it is possible to have P_1, P_2, P_2, P_3 in the trace where P_2 appears twice. In both scenarios above, the assembled stream is always "ABCDEFGHIJ".

Third, there might be a pathological scenario where a packet in the middle is missing. For example, you may only see P_1, P_3 but not P_2 in the trace. This is usually because `tcpdump` misses some packets due to capture buffer overflow. In this project, you do not need to handle this scenario.

Since TCP connections are bi-directional, each TCP connection contains *two* streams: one outgoing stream from client to server (called *uplink*), and the other incoming stream from server to client (called *downlink*). You need to generate them separately. It is possible that an uplink stream, a downlink stream, or both streams of a TCP connection contains no data.

A TCP connection can be closed by either FIN or RESET. A corner case is, additional data may appear after the peer closes the connection, for example:

Packet P_1: Sequence number = 100, Downlink Data = "ABC"
Packet P_2: Sequence number = 103, Downlink Data = "DEFGH"
Uplink RESET or FIN packet closing the connection
Packet P_3: Sequence number = 108, Data = "IJ"

In the above case, P_3 will not be delivered to the receiver application, but it still needs to be included in the assembled stream.

Next, we describe the requirements of this task. Your program reads a PCAP file from `stdin`. The output to `stdout` consists of two parts: the textual part and the binary part.

The textual part consists of n lines ($n \geq 0$), each corresponding to a TCP connection. Each line has exactly six fields separated by space, and the lines need to be sorted alphabetically from small to large (*i.e.,* treat each line as a string, and line x precedes line y if and only if `strcmp(x,y)<0`). The meanings of the six fields are listed below.

1. The client IP address;
2. The client port number;
3. The server IP address;
4. The server port number (always 80);
5. The number of bytes of the uplink stream;
6. The number of bytes of the downlink stream.

The first four fields (*i.e.,* a four-tuple) uniquely represent a TCP connection. You only need to list TCP connections whose either (source or destination) port number is 80, which is the port number

of an HTTP server. For example, assume a connection has the following four-tuple:

(IP1 = 192.168.0.5, Port1 = 45226, IP2 = 157.166.226.25, Port2 = 80)

In the above example, IP1/Port1 is the client-side IP/Port, and IP2/Port2 is the server-side IP/Port. Therefore, the uplink stream is (IP1, Port1)→(IP2, Port2), and the downlink stream is (IP2, Port2)→(IP1, Port1). You can assume that there is no such a connection whose both ports are 80.

After the textual part comes the binary part where you need to output the actual stream data. The binary part consists of 2*n data blocks where the $(2k-1)$-th block ($k \geq 1$) is the uplink stream's data of the k-th connection listed in the textual part, and the $2k$-th block is the downlink stream's data of the k-th connection. There is no separator after a data block (do not insert a newline), and a data block can be empty.

Below is an example of a possible output.

```
192.168.0.5␣45226␣157.166.226.25␣80␣100␣0\n
192.168.0.5␣45227␣157.166.226.25␣80␣80␣500\n
192.168.0.5␣36100␣129.79.78.188␣80␣65␣10000\n
[100 bytes of Connection 1's uplink data][80 bytes of Connection 2's uplink data][500
bytes of Connection 2's downlink data][65 bytes of Connection 3's uplink data][10000
bytes of Connection 3's downlink data]
```

Task 3: Extract HTTP Conversations

The goal of this task is to reconstruct HTTP conversations from the assembled TCP flows. As the key protocol supporting the World Wide Web (WWW), HTTP assumes the client-server computing model, and functions as a request-response protocol. The client, such as a web browser, sends an *HTTP request* message to the server asking for a particular resource object (*e.g.,* an HTML page or an image). The server then returns with an *HTTP response* containing the object data. HTTP runs above TCP, which ensures reliable and in-order delivery of the underlying byte stream over the network.

In this project we focus on the HTTP/1.1 protocol. For simplicity, you can assume that HTTP always uses server port 80, and its TCP connections contain one or more request-response pairs, as shown in Figure 1.

The formats of HTTP/1.1 requests and responses are described below.

An HTTP request consists of a header and an optional body. The textual header consists of several lines separated by `\r\n` (`0x0d 0x0a`, different from the Linux line separator that is `0x0a`). The first line of a request always has the following format (all HTTP/1.1 key words are case insensitive):

```
[Method]␣[URL]␣HTTP/1.1
```

`[Method]` is the request method that can be one of the following: `HEAD`, `GET`, `POST`, `PUT`, and `DELETE`.

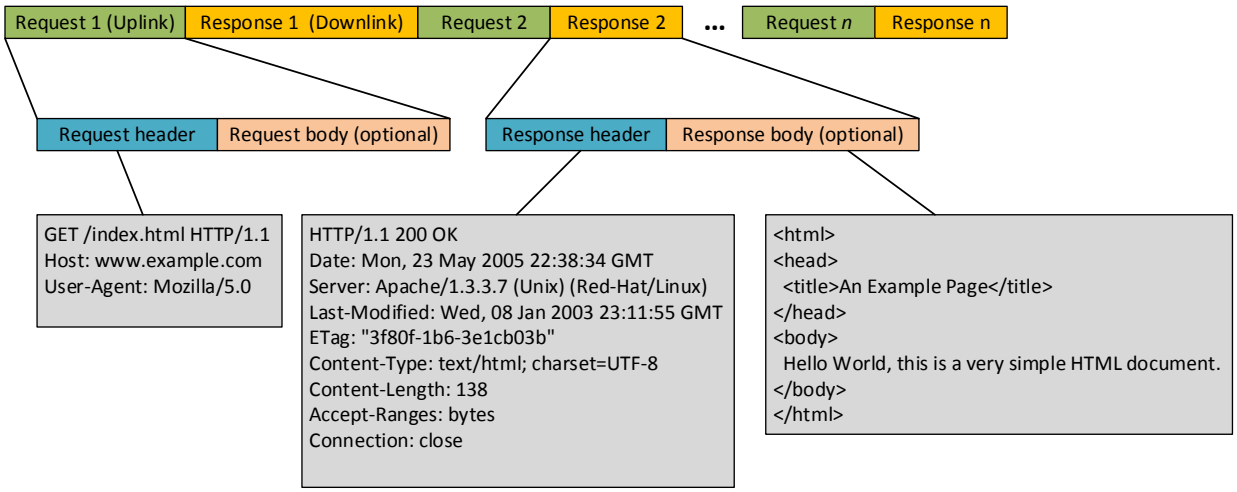

Figure 1: An illustration of HTTP/1.1 traffic pattern.

The most common methods are `GET` (when the browser requests for a resource) and `POST` (when the browser uploads user data). `[URL]` is the request URL.

Subsequently, each of the next several lines contains a *header field* that specifies certain property of the request. For example, the `Host` header field contains the host (domain) name of the requested resource, such as:

```
Host: www.iu.edu
```

The `Content-Length` field contains the request body length in bytes, such as:

```
Content-Length: 2048
```

The `Transfer-Encoding` field dictates that chunked encoding is used (to be described soon):

```
Transfer-Encoding: chunked
```

Your program only needs to parse the above fields (if any of them is present). The request header is terminated by an empty line containing only `\r\n`.

Requests of some types (*e.g.,* `POST`) may contain a body, whose presence is signaled by either the `Content-Length` or the `Transfer-Encoding` field. Your program can just treat the body as an opaque data block in this task.

Next we describe the format of HTTP responses. Similar to an HTTP request, an HTTP response contains a response header and an optional response body. The textual response header also consists of several lines separated by `\r\n` (`0x0d 0x0a`). The first line conforms to the following format:

```
HTTP/1.1 [Code] [Description]
```

`[Code]` is a 3-digit status code and `[Description]` is its textual description. Most frequently used

codes include "`200 OK`", "`404 Not Found`", and "`301 Moved Permanently`" (used for redirection).

Similar to a request header, in a response header, after the first line, each line contains a header field of the response. Your program needs to recognize two header fields: `Content-Length` and `Transfer-Encoding`. The response header is also terminated by a line containing only `\r\n`.

Many responses contain a body, whose presence is signaled by either the `Content-Length` or the `Transfer-Encoding` header field. One exception, however, is that all responses to the `HEAD` request and all 1xx, 204, and 304 responses do not include a body regardless of whether `Content-Length` or `Transfer-Encoding` is present. Your program can just treat the response body as opaque data in this task.

As said, there are two ways to encode a body. When `Content-Length` is given in a request or response where a body is allowed, its field value (0 is allowed) will exactly match the number of bytes in the body. When `Transfer-Encoding:␣chunked` is present, the body uses chunked encoding, which is usually used when the client or the server does not know the size of a resource (*e.g.,* a dynamically-generated one) before starting transmitting the resource.

In chunked encoding, a body consists of zero or more chunks followed by a special termination chunk. Each chunk consists of four parts: (1) the number of bytes of the data contained in the chunk, expressed as a hexadecimal number (the number is always greater than 0); (2) `\r\n`; (3) the actual chunk data; and (4) `\r\n`. The termination chunk, which is always the last chunk, has a chunk size of 0. Below shows an example of encoded data (from Wikipedia):

```
4\r\n
Wiki\r\n
5\r\n
pedia\r\n
e\r\n
␣in\r\n\r\nchunks.\r\n
0\r\n
\r\n
```

Its decoded data is:

```
Wikipedia␣in\r\n
\r\n
chunks.
```

If both `Content-Length` and `Transfer-Encoding` are present in a request or response header, ignore the former. If neither header field is present, then the body length should be assumed to be zero.

Your program needs to handle both persistent HTTP connection and non-persistent HTTP connection. Their difference is, as we discussed in the class, a non-persistent connection carries only one HTTP transaction (*i.e.,* request-response pair), while a persistent connection can carry multiple HTTP transactions sequentially, as illustrated in Figure 1.

We now specify your task. Your program reads a PCAP file from `stdin`, and outputs several lines to `stdout`. Each line corresponds to an HTTP transaction (*i.e.,* a request/response pair), and consists of four fields:

1. The requested URL (the second string in the HTTP request line) in lowercase.
2. The host name (the `Host` field value) in lowercase. If it does not exist, output "`n/a`" (without quote).
3. The 3-digit response code.
4. The response body length in decimal. If `Content-Length` is present, the body length is its field value. If chunked encoding is used, it is the decoded body length (*e.g.,* 23 in the above example). If the response does not have a body, output 0.

For the HTTP transaction example shown in Figure 1, its output line is:

`/index.html␣www.example.com␣200␣138\n`

The HTTP transactions your program outputs must be sorted by the reception time of requests, in an ascending order. That is, transaction x precedes transaction y if and only if the first packet containing x's request occurs before the first packet containing y's request in the trace. Also, your code only needs to process TCP connections whose server port number is 80.

Task 4: Extract Web Resources

This task leads you to the final goal of this project: extract images downloaded from web servers.

So far, you already have the HTTP transaction list computed in Task 3. Now you simply go through the list and pick those transactions that (1) have "`200 OK`" response code, and (2) have request URLs ending with `.jpeg`, `.jpg`, `.png`, `.gif`, or `.webp` (WebP is a new web image format developed by Google, see `https://developers.google.com/speed/webp/`). Note file extensions are case insensitive (*e.g.,* both `.jpg` and `.JPG` are image extensions). For each of these responses, the response body is the image data we want. If you write the data to a file with the same extension, you should be able to see the image using an image viewer.

Your program reads a PCAP file from `stdin`, and outputs the extracted image data to `stdout`. Use the chunked encoding as the output format, with each chunk corresponding to an image. Use lower cases for hexadecimal numbers of chunk size. Also do not forget to include the termination chunk (`0\r\n\r\n`). If the trace does not contain any image, only output the termination chunk. For example, if the trace contains three images whose sizes are 1000 bytes, 2000 bytes, and 3000 bytes, respectively, then the output is:

```
3e8\r\n
[1000 bytes of Image 1's data]\r\n
7d0\r\n
[2000 bytes of Image 2's data]\r\n
bb8\r\n
[3000 bytes of Image 3's data]\r\n
```

```
0\r\n
\r\n
```

The images must appear in the same order as they appeared in the HTTP transaction list generated in Task 3.

Although uncommon, some images may be compressed, as indicated by the `Content-Encoding` field in the response header, such as:

```
Content-Encoding: gzip
```

You will not be able to view such images unless you decompress them. You do not need to perform decompression in this task *i.e.,* just output the compressed image. However, you do need to decode the chunked encoding if it is used.

It is worth mentioning that HTTP is a complex protocol. For simplicity, in this project you are not required to parse advanced protocol elements (byte-range request, HTTP pipelining, *etc.*). Also, you are not required to handle errors. Therefore, it is possible that your program does not produce desired output when being tested against PCAP traces captured by yourself. Also, if the website uses HTTPS (usually using TCP port 443), it is inherently impossible to extract plain text data from the trace unless you have the servers' private key (this is exactly the reason why we need HTTPS). Nevertheless, as long as you precisely follow all given instructions, your code will pass all test cases designed for this project.

Submit Your Program

Compress all your source files (`*.c;*.cpp;*.h;*.java`) into a file in `.zip`, `.bz2`, or `.tar.gz` format.

For example, the following command creates your submission (assuming it's written in C++) called `project2.tar.gz`:

```
tar -cvzf project2.tar.gz file1.cpp file2.cpp file3.cpp file1.h file2.h file3.h
```

where `file*.cpp` and `file*.h` are your C++ source and header files, respectively. You must double check your submission to make sure it includes all files. Take C++ as an example. To compile your program in Linux, type:

```
g++ file1.cpp file2.cpp file3.cpp
```

It will generate an executable file named `a.out`. Do not include the executable in your submission. Also, you are not allowed to use any 3rd-party library. Similar to Project 1, your program can only use built-in C/C++/Java libraries, including the C++ Standard Template Library (STL).

Your program takes a single argument of the task number (1 to 4). For example, the following command runs Task 3 (assuming the compiled executable is `a.out`):

```
./a.out 3
```

The parameter format for C and Java programs is the same as the above. If you use C/C++, we strongly recommend you work on Linux (*e.g.,* Ubuntu or Fedora) with an update-to-date GNU C/C++ compiler (gcc/g++ version 4.3 and above). Do not use other C/C++ compilers that might be incompatible to gcc/g++. For Java, please use JDK version 7.0 or above.

Test Your Program

We use an automated system to test your program, so you need to make sure your output strictly conforms to the required format (*e.g.,* no additional white spaces, no extra empty lines). The output is case-sensitive.

The test system uses input/output redirection for automation. For example, it can issue the following command to force your program to read from `testcase01.pcap`, and write to `testcase01.out`. You can also use this to test your code.

```
./a.out 3 < testcase01.pcap > testcase01.out
```

However, in your program, you must read only from standard input (`stdin`) and write only to standard output (`stdout`). Failure to do so will force us to manually test your program, and you will lose half of the points because of that.

To help you ensure the correct output format, we provide several sample test cases. You can compare your output and the sample output using `diff`:

```
diff [your_output_file] [sample_output_file]
```

If they are identical, `diff` outputs nothing, otherwise it will display the difference. In the latter case, your implementation must be wrong, because given an input PCAP trace, the correct output of all four tasks is unique.

The running time of your program will be measured. You have 1 second for each test case.

	# test cases	Points per test case	Total points
Task 1	3	6	18
Task 2	4	8	32
Task 3	5	8	40
Task 4	2	5	10
Total	100 Points		