HTTPflow: An Introduction

Lev Grevnin and Brian D. Davison
Computer Science & Engineering Dept.
Lehigh University
19 Memorial Drive West
Bethlehem, PA 18015 USA
davison@cse.lehigh.edu

May 2002

Abstract

Internet performance measurement is becoming increasingly important. As more computers join this global network, downloads, gaming, online presentations and even simple messaging may all experience considerable lag times and communication errors. The current work breaks off the piece of the bigger problem and attempts to take a peek into the performance of the HTTP protocol, the driving force behind Web browsing. Being able to analyze HTTP streams for performance can provide a glimpse into this problem for the network under investigation. This paper introduces HTTPflow which captures packets and extracts HTTP-specific information on a per-TCP flow basis. HTTPflow examines all port 80 traffic and extracts HTTP headers and timings for requests and responses.

1 Introduction

The modern Internet is an intricate and sophisticated system. A huge number of messages travel across this world-wide network every second. With this kind of "electronic traffic" certain problems may arise related to its performance. A great variety of network-snooping utilities have been devised to monitor the flow of packets. Packages such as *tcpdump* [10] or *ethereal* [3] reliably capture various types of packets and display their content as well as statistical information. Nevertheless, there is a lack of a public tool which is dedicated to specifically processing HTTP [7] traffic. *HTTPflow* is an attempt to devise such a utility to provide some insight into the performance

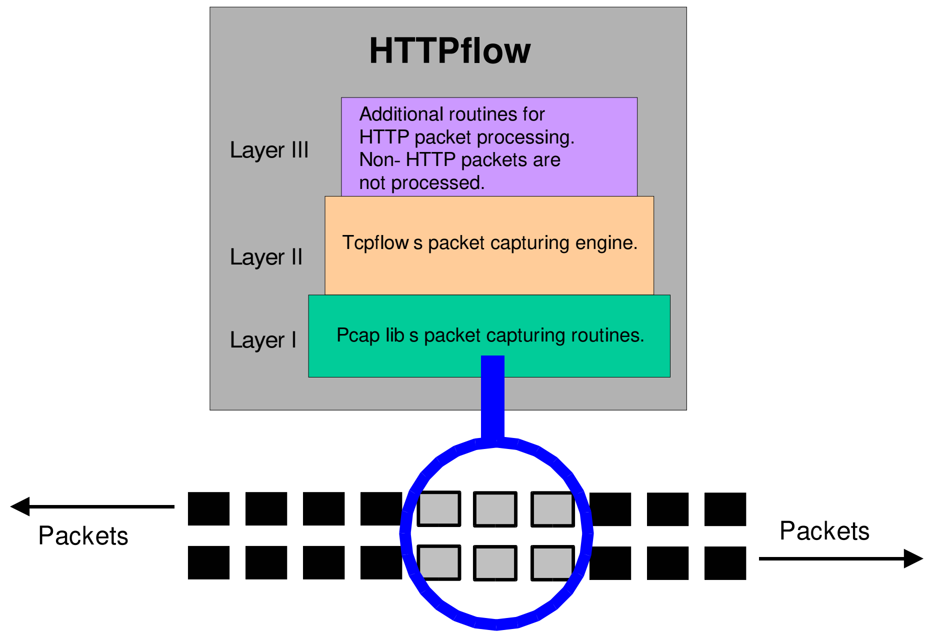

Figure 1: HTTPflow conceptual view.

of the HTTP protocol. This piece of software is based on a modification of a preexisting packet capture utility called *tcpflow* [4], which uses the *pcap* [9] library routines to capture TCP/IP packets.

The reason for this approach is that tcpflow is lightweight (compared to other packet-snooping packages such as ethereal), reliable, well-documented and easy to modify. In addition, it is a good time-saving technique to reuse quality source code and incrementally extend existing work. HTTPflow, like all other packet analyzers, relies on capture of and direct interaction with TCP/IP packets which need to be somehow extracted from the network. Tcpflow, using the powerful pcap library, does this extraction job very well. It is to be noted that one other major advantage of tcpflow is that it reconstructs entire TCP sessions in the logical order, not in packet arrival order. This is quite helpful for reliable interpretation of the HTTP protocol. After tcpflow's routines are used to capture a packet, HTTPflow's routines come in to reconstruct an HTTP message (request or response) based on already captured HTTP packets in addition to the currently processed one. As the process of reconstruction takes place, timestamps are recorded on a per-message basis and sent to the output along with the relevant information about a particular HTTP message.

It is to be noted, however, that the version of tcpflow used (v0.20) is unable to reconstruct fragmented packets. When such packets are detected, the program simply skips them and moves on. In addition, tcpflow doesn't free state information associated with the flows. A bit more coding will have to be done to make tcpflow more efficient.

This paper introduces some of the details of how HTTPflow accomplishes its task as well as talking about how to effectively test the software, run it, and extend it to add more features. A discussion of open issues and deficiencies is also included. The discussion refers to HTTPflow version 0.1 dated February 2002.

2 Background

As mentioned above, the problem is to capture HTTP packets and record timestamps and HTTP headers on a per request or per-response basis. This extends traditional packet analysis where the process ends with the capture of a packet. HTTPflow goes further and attempts to make sense of all the incoming packets representing HTTP messages. When the packets representing a particular request or response have all been picked up by packet capturing routines and processed by HTTP processing routines, the result is sent to the output or a file. With this mechanism in place, a computer running HTTPflow can be placed at the backbone of a major network, like a university network, for traffic monitoring. Outgoing and incoming HTTP traffic is monitored and statistical information can be computed by a post-processor based on the captured data. This information depends on what type of performance parameter is needed to be measured or evaluated. For example, in some cases it would be useful to know the time delays between requests and responses to and from a particular Web site. Regardless of what it is, HTTPflow outputs basic time information or can be extended to output anything else about HTTP messages which can later be used to produce the desired computation.

Similar systems have been developed by others. Gribble [8] cites the use of a customized HTTP module written on top of a packet capture utility. Likewise Wolman *et al.* [14] employed passive network monitoring with a kernel packet filter to capture HTTP (and other) traffic. Wooster *et al.* [15] developed HTTPDump to extract HTTP headers from content collected by tcpdump. More recently, Smith *et al.* [11] reported on the benefits of passive tracing of TCP/IP packet headers for describing Web traffic. Others have created tools to capture particular types of packets, such as streaming and

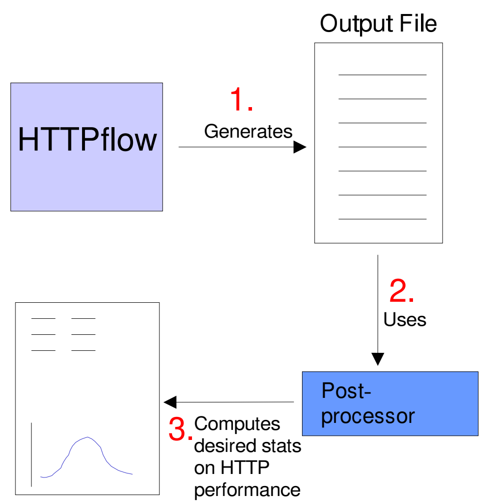

Figure 2: Typical HTTPflow use.

multimedia workloads [13, 2].

The most similar effort to ours, albeit more extensive, is that of Feldmann [5, 6]. In that work, tcpdump has been extended to extract HTTP and TCP/IP information online. The resulting proprietary system, implemented within PacketScope [1], was used to collect trace data from AT&T Labs–Research and AT&T WorldNet for a number of projects spanning multiple years.

3 HTTPflow

While based on tcpflow, HTTPflow introduces a number of changes and features. The most notable and significant of these is a different output. Instead of dumping contents of TCP packets, like tcpflow does, HTTPflow prints out the information about every processed HTTP message in the following format:

```
Source IP address-Destination IP address
time of beginning of headers
headers...
headers...
headers...
\r\n\r\n
```

```
time of end of headers
time of end of body
```

HTTPflow is most commonly run from the command prompt as follows:

```
./HTTPflow -c > out 2> err
```

where -c indicates that the output should go to the console and both stderr and stdout are redirected to their appropriate files. Optionally, a host flag can be used to monitor messages to and from a particular host. For example:

```
./HTTPflow -c host http://abc.com > out 2> err
```

With regard to the error messages, it should be noted that there is a variable called `MY_DEBUG` defined at the beginning of the http.c file which when not commented out, will be used in conjunction with the custom `print()` routine to send debug information.

4 Implementation

During implementation, three important assumptions were made:

- The first packet of a request must contain either one of the following words in full as the first bytes of the data section: GET, HEAD, OPTIONS, DELETE, POST, PUT. The first packet of a response must contain the term HTTP in full. If this condition is not met (which will almost never happen, given the way operating systems and web browsers form HTTP packets), an initial packet in the, say, GET request, containing just "G" as the data, must be stored and its time recorded. If eventually it will turn out to be that this is not a GET request, that information must be cleared. This can turn out to be messy and it is easier to make a simplifying assumption (which is a reasonable one too!).

- Any request or response packet not identified as a part of a currently processed request or response stream or the beginning of the request or response stream will not get processed and will be skipped.

- Given that HTTP requests and responses have similar structure (headers, `\r\n\r\n`, body) their processing is handled in an identical manner: the `process_response()` function in `http.c` is simply a wrapper for the already defined and implemented `process_request()` routine.

The best way to demonstrate the structure of the program and how it accomplishes its task of processing HTTP messages can be demonstrated using an output sample of a processed request.

```
#1:  128.180.098.216.31274-195.230.090.026.00080
#2:  Time: 1015019677 s. 929220 us.
#3:  GET / HTTP/1.0
#4:  Connection: Keep-Alive
#5:  User-Agent: Mozilla/4.78 [en] (X11; U; Linux 2.4.7-10 i586)
#6:  Host: www.lenta.ru
#7:  Accept: image/gif, image/x-xbitmap, image/jpeg, */*
#8:  Accept-Encoding: gzip
#9:  Accept-Language: en
#10: Accept-Charset: iso-8859-1,*,utf-8
#11: Cookie: ruid=0QcAAKT4ajxXQAAAAVdQANC6/
#12:
#13: Time: 1015019677 s. 929220 us.
#14: Time: 1015019677 s. 929220 us.
```

Line #1 is the source-destination IP address combination representing this message's direction. In this case one can see it's a request since the destination is the IP number with port 80. All IP numbers (src, dst) for a given packet can be determined from the `flow_t` structure defined in the `tcpflow.h` file. Line #2 records the time when the beginning of the headers was detected. The time was passed into the HTTP processing code in the http.c file using the `pac_info` structure defined in the `tcpflow.h` file. The structure was filled with the seconds and microseconds of the packet arrival in the `datalink.c` file, `dl_ethernet()` routine, right before control was handed over to higher-level processing — the `process_ip()` routine.

Lines #3 through #12 are the contents of the headers section of the message. Line #12, of course, is the blank line which indicates the end of the headers section (and is part of the headers). Line #13 is the time when the last packet of the headers came in. Line #14 is the time the body section of the message was over. In our example it turns out to be the same as the end of the headers time. To preserve generality, the body end time is always printed, even if there is no body, in which case, it will be equal to the end of headers time.

When working with HTTPflow, it must be remembered that only HTTP messages are processed. We make the assumption that the only HTTP traffic of interest is on port 80. Thus, to determine whether a packet belongs to

128.180.098.216.61274-195.230.090.026.00080	GET /file.txt
145.231.072.211.00080-211.234.012.023.34257	HTTP/1.1
Status_of222.012.045.112.00080-231.061.052.032.11314	Expecting_body_bytes_5200
....	....
....	....
....	....

Figure 3: Sample hash table.

an HTTP message or not, one needs to simply look at the IP address combination — the source and the destination IP numbers. If any of the numbers includes port 80, this indicates that the packet is likely to belong to a stream of packets comprising an HTTP request or response. As already indicated, the modifications to tcpflow allow for processing of HTTP requests and responses passing through the machine on which HTTPflow is installed. But how does the code distinguish between packets from different requests or responses? As soon as the first packet of a message comes in, processing of the new stream is started by adding the appropriate IP address combination to a hash table as a key, and as for the value, strings containing headers and times are added incrementally as the packets comprising the message come in, as shown in Figure 3. Eventually, when the entire message is completed, it is sent out to the screen with the `output_result()` routine. Then, the memory is freed from the hash table. In addition, the hash table is also used to store various statuses about requests and responses still in process, such as, remaining body length, how large the chunk is, what the total content length is, etc.

For HTTP requests and responses that use chunk encoding to transmit the message body, special attention must be given. The `process_chunked_encoded()` routine is used to skip through the chunks of the body. The process is very involved and there are a number of helper routines to facilitate the processing (`validate_chunked_body_end_bRbN()`, `validate_chunked_body_end_bN()`, `process_rest_of_chunked_PPpacket()`, `get_chunk_length()`, `convert_hex_to_dec()`, `is_chunk_encoded()`). It should be noted that there were a number of general-purpose routines defined to be used for string manipulation, memory management, accessing data in the hash table. All of these routines are defined in the `http.c` file.

In addition to the already defined output, there may be a number of useful extensions added to it.

- If body size is needed to be sent to the output also, it can be accomplished in several ways: If the message uses chunked encoding, as the packets with body chunks come in, their values are added to the value of the hash table key-value pair with the key being of the format similar to the following: `123.123.445.233.80-333.231.445.213.5536BodySize` and the value consisting of a number of bytes of the sum of chunk sizes of the chunked body. If the message uses content length, the situation is similar — the value is added once, when the Content-Length header is detected. If TCP signals of FIN or RST are used to terminate the connection (and the body), body bytes prior to the termination must be counted up and added to the value in the above-mentioned hash table key-value pair. Eventually, when the end of the body is encountered, the value of this hash table pair must be attached to the value string of the hash table pair with key containing the appropriate stream IP number combination.

- Another important extension to the HTTPflow package would be the inclusion of a post-processor which would take the HTTPflow output file as an input and match up requests and responses in pairs. The likely approach includes setting up an internal hash table based on the requests seen. That is, the key consisting of the request's IP address combination and the value consisting of the rest of the request (headers, arrival times, etc.). As this program moves through the output file, it will enter all the requests in the above manner while looking up the responses after the appropriate requests have already been entered into the table. If a particular response IP address combination matches it's reverse in the hash table (the request), this indicates that this response has an appropriate request in the hash table. Upon this finding, the program will then perform useful computations, such as calculating total latencies, first byte response times, etc.

5 Evaluation

Testing this application at first on a real data stream is not effective — lots of extra information is flowing through and in many cases one has to wait a while before the right combination of packets comes along to test for a particular case. The creation of a very small HTTP client program can greatly simplify the process of testing and fine-tuning HTTPflow's HTTP handling capability. For our testing, we created a program called HTTP-Client, written in Java. It creates a socket to a not-so-popular website http://www.lenta.ru/ (which is what we want, since we only want our request or response to that site to be detected), and sends to it a stream of packets. Running HTTPflow in the following fashion `./HTTPflow -c host www.lenta.ru >output 2> error` will capture the packets to and from www.lenta.ru only and output the result or produce error. We use an array variable of type `String` to contain the bytes sent out in separate packets. For example, the below arrangement contains three packets comprising a syntactically valid POST request sent out to www.lenta.ru.

```
String packets[] = {
 "POST /file.txt Transfer-encoding: chunked \r\n\r\n10\r\n^^",
 "^^^^^^^^^^^^^^\r\n3\r\n^^"                                   ,
 "^\r\n5\r\n^^^^^\r\n2\r\n^^\r\n0\r\n"
};
```

This way, a very controlled, simple and flexible mechanism for testing is in place.

Another important area is checking for memory leaks. A rough estimate can be made by running HTTPflow simultaneously with some system monitoring utility like `top(1)`. Top will capture the amount of memory used by the HTTPflow process. If after some time the memory is growing, this indicates that there is a memory leak somewhere in the application which must be fixed. More complete tests can typically be performed with the appropriate code profiler and/or debugger.

6 Discussion

There are a number of issues left for consideration. One of them deals with the order of requests and responses in the output file. Imagine the following situation: an incorrect request is sent to a server followed by a body. As soon as the server receives `\r\n\r\n` to signify the end of the request headers it starts processing the request. Upon discovering that there is an error in the headers, the server will most likely send out an HTTP error response, without waiting for the rest of the request (the request body) to be received. HTTPflow will capture the response as it is still processing the request and output the response. Eventually, when the entire request is complete, that request will get printed out also. The problem in this scenario lies in the fact that an HTTP response gets printed first, that is, before the HTTP request. This may not seem to be a problem for logging the results, but has to be considered in post-processing operations. The post-processing routine will need to match up every request with every response, as mentioned above.

The other issue deals with inefficiencies in the memory management process. If there are requests or responses which were terminated because the server crashed or for some other abnormal reason, the memory in the hash table will still be occupied and will never be freed because the end of these messages will never come. Considering that hundreds of thousands of requests and responses to and from countless thousands of websites will be passing through HTTPflow, this is not such an unlikely possibility to have some of these messages be "stuck" in the above situation taking up a sizable chunk of RAM. What is needed is a mechanism to time-out the handling of such messages.

Also, it must be remembered that as HTTPflow is spitting out the results of HTTP message processing, the size of the file to which this output is redirected to will become very large. Therefore, a file splitting mechanism is needed. The *split* utility which comes with the UNIX or Linux operating systems does precisely that. split creates output files containing consecutive sections of the input. The sizes of these files can be arbitrarily specified.

In addition to the concerns about tcpflow mentioned in the Introduction, several bugs have been identified in the initial version of HTTPflow.

- The handling of chunked encoded messages works well except for the case when the chunk number is spread over several packets. In this case, the program throws an error and continues on. Nevertheless, the great majority of cases encountered in most HTTP messages is handled correctly.

- The handling of message interruption using FIN or RST packets is implemented but not tested fully and so was commented out. The function responsible for this work is `rst_or_fin_terminated()`. It should be called in the `process_request()` function. Note, that at present we assume that there will never be any real data that is part of a packet with an RST or FIN flag set. The variable indicating that an FIN or RST flag has been set is in the `pac_info` structure. This variable gets set in the `process_tcp()` routine in the `tcpip.c` file.

- There is a small memory leak. It comes from not freeing up the memory when using regular expressions. This code was intentionally left out to keep certain routines simple for the ease of debugging them. Regex documentation [12] must be consulted to determine how to free memory after using a regular expression. The routine `Regfree()` is responsible for this operation.

7 Summary

The Internet has expanded tremendously over the past few years. As a result, evaluation of performance of the HTTP protocol became a problem of interest. HTTPflow is a utility which logs HTTP headers and timestamps. This utility outputs information on a per message basis. After a sufficient amount of information has been collected, the post-processor scans over the output file, matches up requests with appropriate responses and computes the desired performance parameters. The current version of HTTPflow includes the following features:

- Processing of messages with no body (GET, HEAD, DELETE, HTTP response).
- Processing of messages with "Content-length"-specified body (PUT, POST, HTTP response)
- Processing of messages with chunk encoded body (PUT,POST, HTTP response)

The ability to measure HTTP performance will arm the network analyst with the power of quantitative analysis. This, in turn, will no doubt help in understanding what the scope of the problem really is and what needs to be done to alleviate it.

References

[1] N. Anerousis, R. Caceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. K. Ramakrishnan, and J. Rexford. Using the AT&T Labs PacketScope for Internet measurement, design, and performance analysis. In *AT&T Services and Infrastructure Performance Symposium*, Nov. 1997.

[2] M. Chesire, A. Wolman, G. M. Voelker, and H. M. Levy. Measurement and analysis of a streaming media workload. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS-01)*, San Francisco, Mar. 2001.

[3] G. Combs et al. The Ethereal network analyzer. Available from http://www.ethereal.com/, 2002.

[4] J. Elson. tcpflow — a TCP flow recorder. Available from http://www.circlemud.org/~jelson/software/tcpflow/, 2002.

[5] A. Feldmann. Continuous online extraction of HTTP traces from packet traces. In *World Wide Web Consortium Workshop on Web Characterization*, Cambridge, MA, Nov. 1998. Position paper.

[6] A. Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. *WWW9 / Computer Networks*, 33(1-6):321–335, 2000.

[7] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, `http://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[8] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, Dec. 1997.

[9] V. Jacobson, C. Leres, and S. McCanne. Packet capture library. Available from http://www.tcpdump.org/.

[10] V. Jacobson, C. Leres, and S. McCanne. tcpdump. Available from http://www.tcpdump.org/, June 1989.

[11] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP protocol headers can tell us about the Web. In *Proceedings of the ACM SIGMETRICS/Performance Conference on Measurement and Modeling of Computer Systems*, pages 245–256, 2001.

[12] R. Stallman et al. Gnu regex library. Available from http://www.gnu.org/directory/regex.html, 1993.

[13] J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan. Mmdump — A tool for monitoring Internet multimedia traffic, Oct. 2000.

[14] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. R. Karlin, and H. M. Levy. Organization-based analysis of web-object sharing and caching. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[15] R. O. Wooster, S. Williams, and P. Brooks. HTTPDUMP: Network HTTP packet snooper. Working paper available at `http://www.cs.vt.edu/~chitra/work.html`, Apr. 1996.