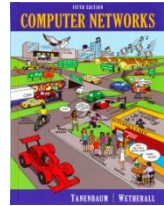


Lab Exercise – HTTP



Objective

HTTP (HyperText Transfer Protocol) is the main protocol underlying the Web. The trace file is here: <http://scisweb.ulster.ac.uk/~kevin/com320/labs/wireshark/trace-http.pcap>

Step 1: Capture a Trace

Capture a trace of your browser making HTTP requests as follows; alternatively, you may use a supplied trace. Now that we seen how a GET works, we will observe your browser as it makes HTTP requests. Browser behavior can be quite complex, using more HTTP features than the basic exchange, so we will set up a simple scenario. We are assuming that your browser will use HTTP in this simple scenario rather than newer Web protocols such as SPDY, and if this is not the case you will need to disable SPDY.

1. Use your browser to find two URLs with which to experiment, both of which are HTTP (not HTTPS) URLs with no special port. The first URL should be that of a small to medium-sized image, whether .jpg, .gif, or .png. We want some static content without embedded resources. You can often find such a URL by right-clicking on unlinked images in web pages to tell your browser to open the URL of the image directly. The second URL should be the home page of some major web site that you would like to study. It will be complex by comparison. Visit both URLs to check that they work, then keep them handy outside of the browser so you can cut-and-paste them.
2. Prepare your browser by reducing HTTP activity and clearing the cache. Apart from one fresh tab that you will use, close all other tabs, windows to minimize HTTP traffic.
3. Launch Wireshark and start a capture with a filter of “tcp port 80”. We use this filter because there is no shorthand for HTTP, but HTTP is normally carried on TCP port 80.

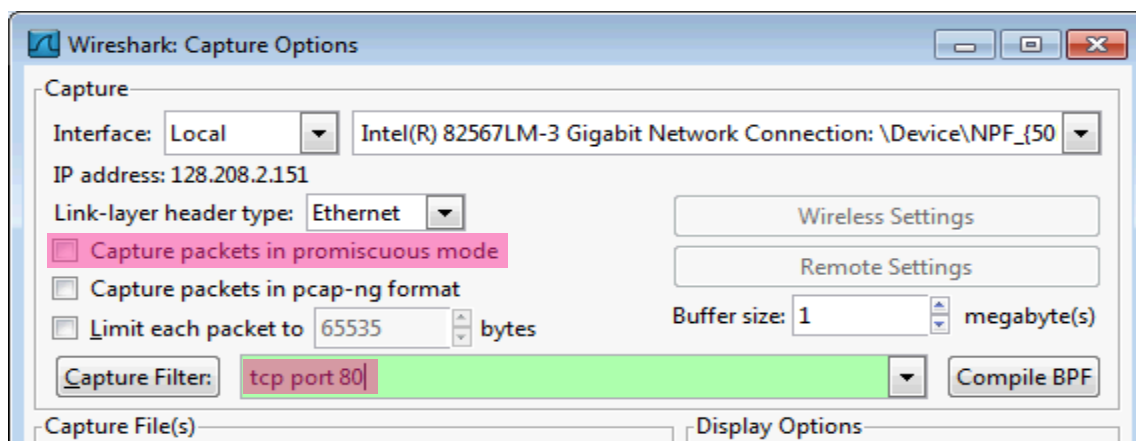


Figure 2: Setting up the capture options

4. *Fetch the following sequence of URLs, after you wait for a moment to check that there is no HTTP traffic.* If there is HTTP traffic then you need to find and close the application that is causing it. Otherwise your trace will have too much HTTP traffic for you to understand. You will paste each URL into the browser URL bar and press Enter to fetch it. Do not type the URL, as this may cause the browser to generate additional HTTP requests as it tries to auto-complete your typing.
 - a. *Fetch the first static image URL by pasting the URL into the browser bar and pressing “Enter” or whatever is required to run your browser.*
 - b. *Wait 10 seconds, and re-fetch the static image URL.* Do this in the same manner, and without using the “Reload” button of your browser, lest it trigger other behavior.
 - c. *Wait another 10 seconds, and fetch the second home page URL.*
5. *Stop the capture after the fetches are complete.* You should have a window full of trace in which the protocol of some packets is listed as HTTP – if you do not have any HTTP packets there is a problem with the setup such as your browser using SPDY instead of HTTP to fetch web pages.

Figure 3: Trace of HTTP traffic showing the details of the HTTP header

Step 2: Inspect the Trace

To focus on HTTP traffic, enter and apply a filter expression of “http”. This filter will show HTTP requests and responses, but not the individual packets that are involved. Recall that an HTTP response carrying content will normally be spread across multiple packets. When the last packet in the response arrives, Wireshark assembles the complete response and tags the packet with protocol HTTP. The earlier packets are simply TCP segments carrying data; the last packet tagged HTTP includes a list of all the earlier packets used to make the response. A similar process occurs for the request, but in this case it is common for a request to fit in a single packet. With the filter expression of “http” we will hide the intermediate TCP packets and see only the HTTP requests and responses. With this filter, your Wireshark display should be similar to the figure showing our example.

Select the first GET in the trace, and expand its HTTP block. This will let us inspect the details of an HTTP request. Observe that the HTTP header follows the TCP and IP headers, as HTTP is an application protocol that is transported using TCP/IP. To view it, select the packet, find the HTTP block in the middle panel, and expand it (by using the “+” expander or icon). This block is expanded in our figure.

Explore the headers that are sent along with the request. First, you will see the GET method at the start of the request, including details such as the path. Then you will see a series of headers in the form of tagged parameters. There may be many headers, and the choice of headers and their values vary from browser to browser. See if you have any of these common headers:

- Host. A mandatory header, it identifies the name (and port) of the server.
- User-Agent. The kind of browser and its capabilities.
- Accept, Accept-Encoding, Accept-Charset, Accept-Language. Descriptions of the formats that will be accepted in the response, e.g., text/html, including its encoding, e.g., gzip, and language.
- Cookie. The name and value of cookies the browser holds for the website.
- Cache-Control. Information about how the response can be cached.

The request information is sent in a simple text and line-based format. If you look in the bottom panel you can read much of the request directly from the packet itself!

Select the response that corresponds to the first GET in the trace, and expand its HTTP block. The Info for this packet will indicate “200 OK” in the case of a normal, successful transfer. You will see that the response is similar to the request, with a series of headers that follow the “200 OK” status code. However, different headers will be used, and the headers will be followed by the requested content. See if you have any of these common headers:

- Server. The kind of server and its capabilities.
- Date, Last-Modified. The time of the response and the time the content last changed.
- Cache-Control, Expires, Etag. Information about how the response can be cached.

Answer the following questions:

1. *What is the format of a header line? Give a simple description that fits the headers you see.*
2. *What headers are used to indicate the kind and length of content that is returned in a response?*

Step 3: Content Caching

The second fetch in the trace should be a re-fetch of the first URL. This fetch presents an opportunity for us to look at caching in action, since it is highly likely that the image or document has not changed and therefore does not need to be downloaded again. HTTP caching mechanisms should identify this opportunity. We will now see how they work.

Select the GET that is a re-fetch of the first GET, and expand its HTTP block. Likely, this will be the second GET in the trace. However, look carefully because your browser may issue other HTTP requests for its own reasons. For example, you might see a GET for /favicon.ico in the trace. This is the browser requesting the icon for the site to use as part of the browser display. Similarly, if you typed in the URL bar your browser may have issued GETs as part of its auto-completion routine. We are not interested in this background browser activity at the moment.

Now find the header that will let the server work out whether it needs to send fresh content. We will ask you about this header shortly. The server will need to send fresh content only if the content has changed since the browser last downloaded it. To work this out, the browser includes a timestamp taken from the previous download for the content that it has cached. This header was not present on the first GET since we cleared the browser cache so the browser had no previous download of the content that it could use. In most other respects, this request will be the same as the first time request.

Finally, select the response to the re-fetch, and expand its HTTP block. Assuming that caching worked as expected, this response will not contain the content. Instead, the status code of the response will be “304 Not Modified”. This tells the browser that the content is unchanged from its previous copy, and the cached content can then be displayed.

Answer the following questions :

1. *What is the name of the header the browser sends to let the server work out whether to send fresh content?*
2. *Where exactly does the timestamp value carried by the header come from?*

Step 4: Complex Pages

Now let's examine the third fetch at the end of the trace. This fetch was for a more complex web page that will likely have embedded resources. So the browser will download the initial HTML plus all of the embedded resources needed to render the page, plus other resources that are requested during the execution of page scripts. As we will see, a single page can involve many GETs!

To summarize the GETs for the third page, bring up a HTTP Load Distribution panel. You will find this panel under "Statistics" and "HTTP". You can filter for the packets that are part of the third fetch by removing the packets from the earlier part of the trace by either time or number. For example, use "frame.number>27" or "frame.time_relative>24" for our trace.

Looking at this panel will tell you how many requests were made to which servers. Chances are that your fetch will request content from other servers you might not have suspected to build the page. These other servers may include third parties such as content distribution networks, ad networks, and analytics networks. Our panel is shown below – the page fetch involved 95 requests to 4 different servers!

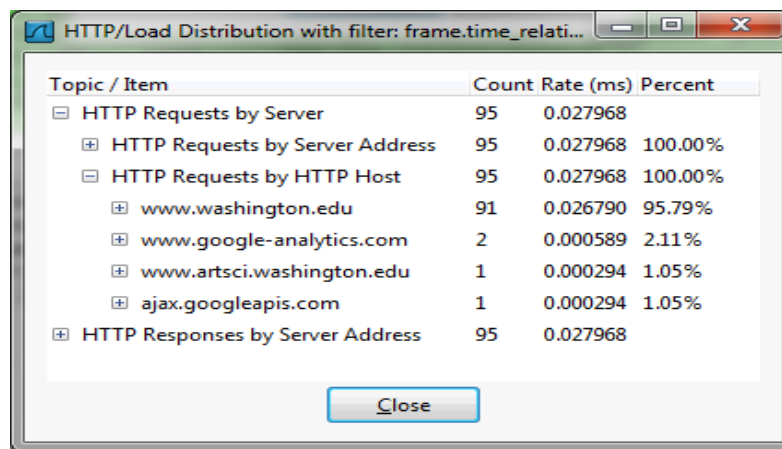
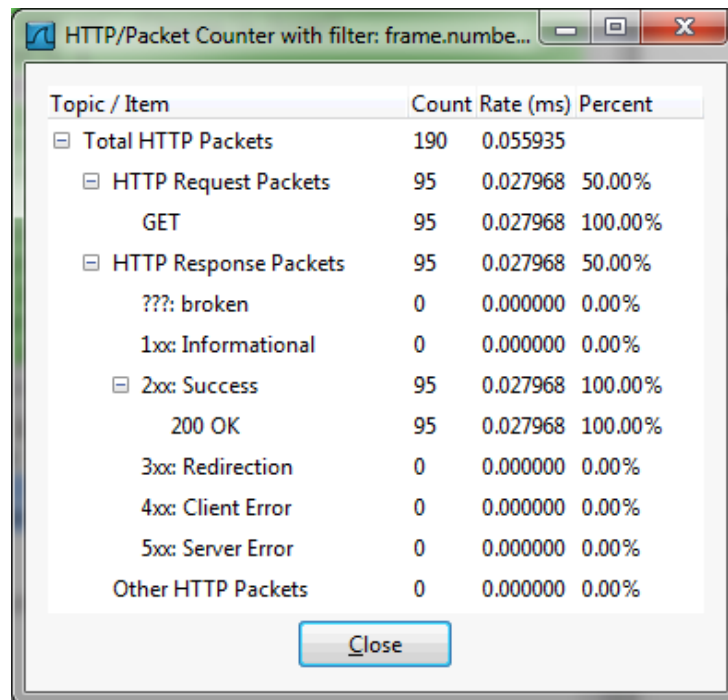


Figure 4: HTTP Load Distribution panel

For a different kind of summary of the GETs, bring up a HTTP Packet Counter panel. You will also find this panel under "Statistics" and "HTTP", and you should filter for the packets that are part of the third fetch as before. This panel will tell you the kinds of request and responses. Our panel is shown in the figure below. You can see that it consists entirely of GET requests that are matched by 200 OK responses. However, there are a variety of other response codes that you might observe in your trace, such as when the resource is already cached.



The screenshot shows a window titled "HTTP/Packet Counter with filter: frame.numbe...". It contains a table with the following data:

Topic / Item	Count	Rate (ms)	Percent
Total HTTP Packets	190	0.055935	
HTTP Request Packets	95	0.027968	50.00%
GET	95	0.027968	100.00%
HTTP Response Packets	95	0.027968	50.00%
??? : broken	0	0.000000	0.00%
1xx: Informational	0	0.000000	0.00%
2xx: Success	95	0.027968	100.00%
200 OK	95	0.027968	100.00%
3xx: Redirection	0	0.000000	0.00%
4xx: Client Error	0	0.000000	0.00%
5xx: Server Error	0	0.000000	0.00%
Other HTTP Packets	0	0.000000	0.00%

A "Close" button is located at the bottom right of the window.

Figure 5: HTTP Packet Counter panel

You might be curious to know what content is being downloaded by all these requests. As well as seeing the URLs in the Info column, you can get a summary of the URLs in a HTTP Request panel under “Statistics” and “HTTP”. Each of the individual requests and responses has the same form we saw in an earlier step. Collectively, they are performed in the process of fetching a complete page with a given URL.

For a more detailed look at the overall page load process, use a site such as Google’s PageSpeed or webpagetest.org. These sites will test a URL of your choice and generate a report of the page load activity, telling what requests were fetched at what times and giving tips for decreasing the overall page load time. We have shown the beginning of the “waterfall” diagram for the page load corresponding to our trace in the figure below. After the initial HTML resource is fetched there are many subsequent quick fetches for embedded resources such as JavaScript scripts, CSS stylesheets, images, and more.

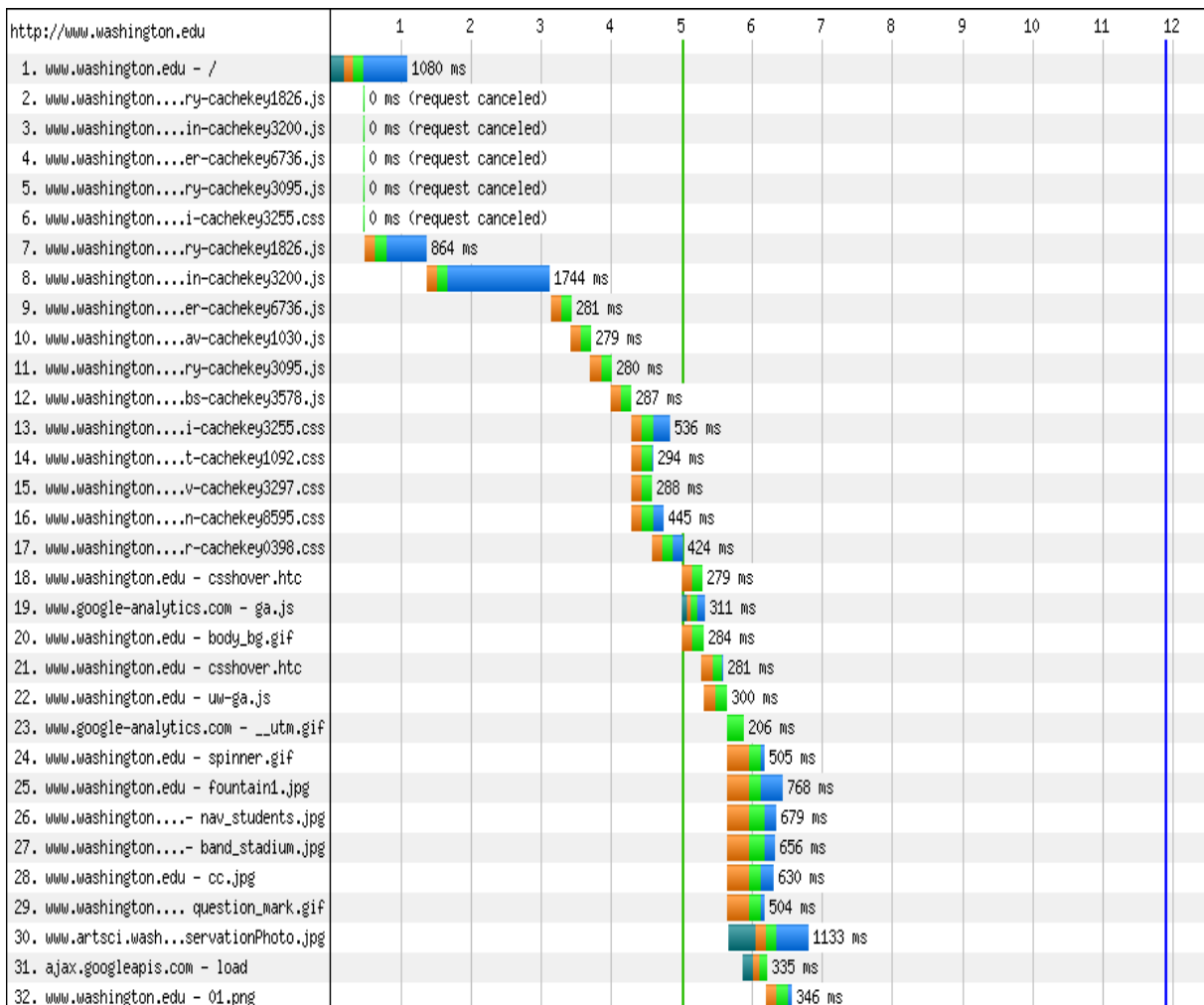


Figure 6: Start of waterfall graph for www.washington.edu (from pageloadtest.org)

Homework - Explore Your Network

Explore HTTP on your own once you have finished this lab. Some suggestions:

- Study how web pages lead to a pattern of HTTP requests. Many popular web sites have relatively complex pages that require many HTTP requests to build. Moreover, these pages may continue to issue “asynchronous” HTTP requests once they appear to have loaded, to load interactive displays or prepare for the next page, etc. You will see this activity when you find HTTP requests that continue after a page is loaded.
- Look at video streaming HTTP traffic. We have looked at web HTTP traffic, but other applications make HTTP requests too. It is common for streaming video clients embedded in browsers like Netflix to download content using a HTTP fetches of many small “chunks” of video. If you look at other applications, you may find that many of them use HTTP to shift about content, though often on a port different than port 80.