# Lab Exercise – HTTP

## Objective

HTTP (HyperText Transfer Protocol) is the main protocol underlying the Web.

## Requirements

**Wireshark**: This lab uses Wireshark to capture or examine a packet trace. A packet trace is a record of traffic at some location on the network, as if a snapshot was taken of all the bits that passed across a particular wire. The packet trace records a timestamp for each packet, along with the bits that make up the packet, from the low-layer headers to the higher-layer contents. Wireshark runs on most operating systems, including Windows, Mac and Linux. It provides a graphical UI that shows the sequence of packets and the meaning of the bits when interpreted as protocol headers and data. The packets are color-coded to convey their meaning, and Wireshark includes various ways to filter and analyze them to let you investigate different aspects of behavior. It is widely used to troubleshoot networks. You can download Wireshark from www.wireshark.org if it is not already installed on your computer. We highly recommend that you watch the short, 5 minute video "Introduction to Wireshark" that is on the site.

**telnet**: This lab uses telnet to set up an interactive two-way connection to a remote computer. telnet is installed on Window, Linux and Mac operating systems. It may need to be enabled under Windows. Select "Control Panel" and "More Settings" (Windows 8) or "Programs and Features" (Windows 7), then "Turn Windows Features on or off". From the list that is displayed, make sure that "Telnet Client" is checked. If you cannot see the text you type when in a telnet session, you may need to use a telnet command to set the "local echo" variable. Alternatively, if you are having difficulty enabling or using Windows telnet, you may install the PuTTY client which uses a GUI to launch a telnet session.

**Browser**: This lab uses a web browser to find or fetch pages as a workload. Any web browser will do.

## Step 1: Manual GET with Telnet

*Use your browser to find a reasonably simple web page with a short URL, making sure it is a plain HTTP URL with no special port number.* Since HTTP is a text-based application protocol, we can see how it works by entering our own HTTP requests and inspecting the HTTP responses. To do this you will use telnet in the place of a web browser, using the URL you select as a test case. You might a top level page of your school web server, e.g., http://www.mit.edu/index.html.

*Divide the URL into the server name, and the path portion*, e.g., www.mit.edu and "/index.html". If your URL ends with a "/" then the path portion will be "/". Or it may be that the path is really "/index.html" and the browser and web server are performing the translation for you. To check if this is the real URL, enter the URL with /index.html at the end into your browser and see if it works.

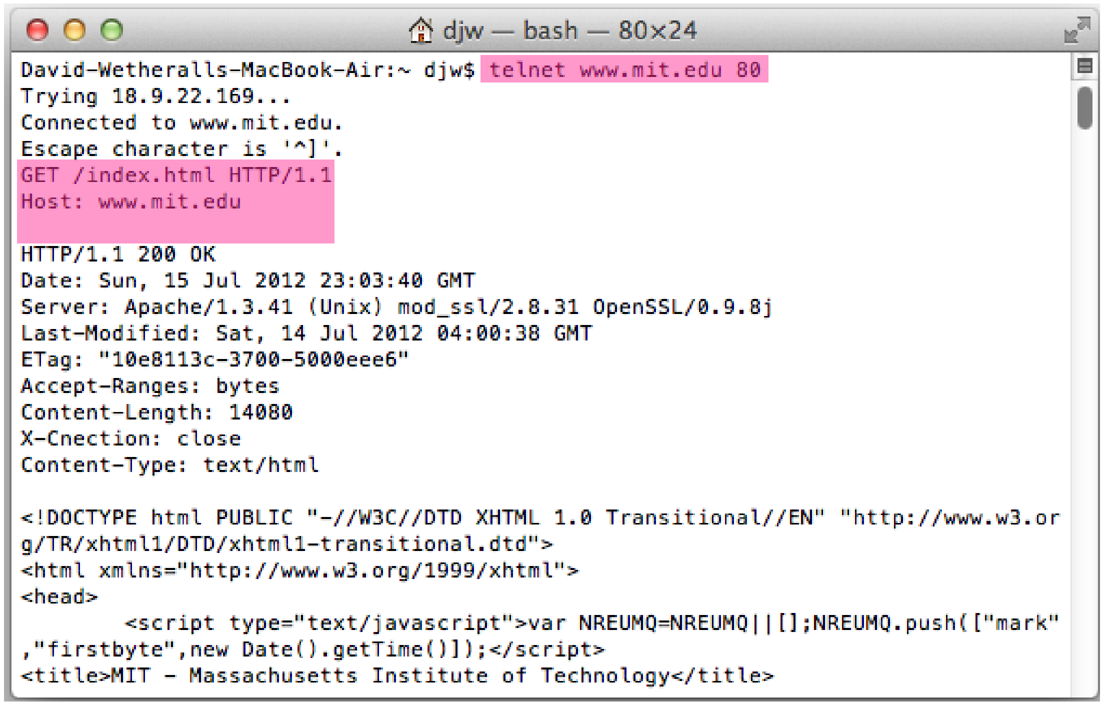
*Use telnet to fetch the page.* What you will do is telnet to port 80 on the server, the standard HTTP port, and then issue HTTP commands acting as the browser. Proceed as follows:

1. *Run telnet and connect to server on port 80.* You can do this from a terminal or command prompt by issuing a command such as "`telnet www.mit.edu 80`". Or if you are using putty to telnet, fill in the server and port on the configuration screen, and select "telnet" and "Never" close window.
2. *Once you are connected, issue an HTTP GET command by typing the three lines below.* The first two lines identify the path and server. The last line is a blank line, to tell the server there are no more headers. It is easily missed, but it is mandatory.

```
GET /index.html HTTP/1.1
Host: www.mit.edu
```

3. *Observe the response that comes back.* If the connection does not close by itself, you may close it by typing the telnet escape character of "control-]" and then typing the command "q" for quit.

Congratulations, you have issued your own GET and seen the inner workings of the web! Our interaction is shown in the figure below, with the parts that we typed highlighted. You may need to scroll back up to see the beginning of your interaction. The details of your output will vary, but they should take the basic form of a web interaction between your browser and a server: a command followed by various client request headers, then the server response, first with a status code and header information, and then with the requested document itself. If the status code is not a "200 OK" then something is wrong. Your command syntax may have an error in it, or an incorrect URL may be the problem.



```
David-Wetheralls-MacBook-Air:~ djw$ telnet www.mit.edu 80
Trying 18.9.22.169...
Connected to www.mit.edu.
Escape character is '^]'.
GET /index.html HTTP/1.1
Host: www.mit.edu

HTTP/1.1 200 OK
Date: Sun, 15 Jul 2012 23:03:40 GMT
Server: Apache/1.3.41 (Unix) mod_ssl/2.8.31 OpenSSL/0.9.8j
Last-Modified: Sat, 14 Jul 2012 04:00:38 GMT
ETag: "10e8113c-3700-5000eee6"
Accept-Ranges: bytes
Content-Length: 14080
X-Cnection: close
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.or
g/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <script type="text/javascript">var NREUMQ=NREUMQ||[];NREUMQ.push(["mark"
,"firstbyte",new Date().getTime()]);</script>
<title>MIT - Massachusetts Institute of Technology</title>
```

Figure 1: Performing an HTTP GET with telnet

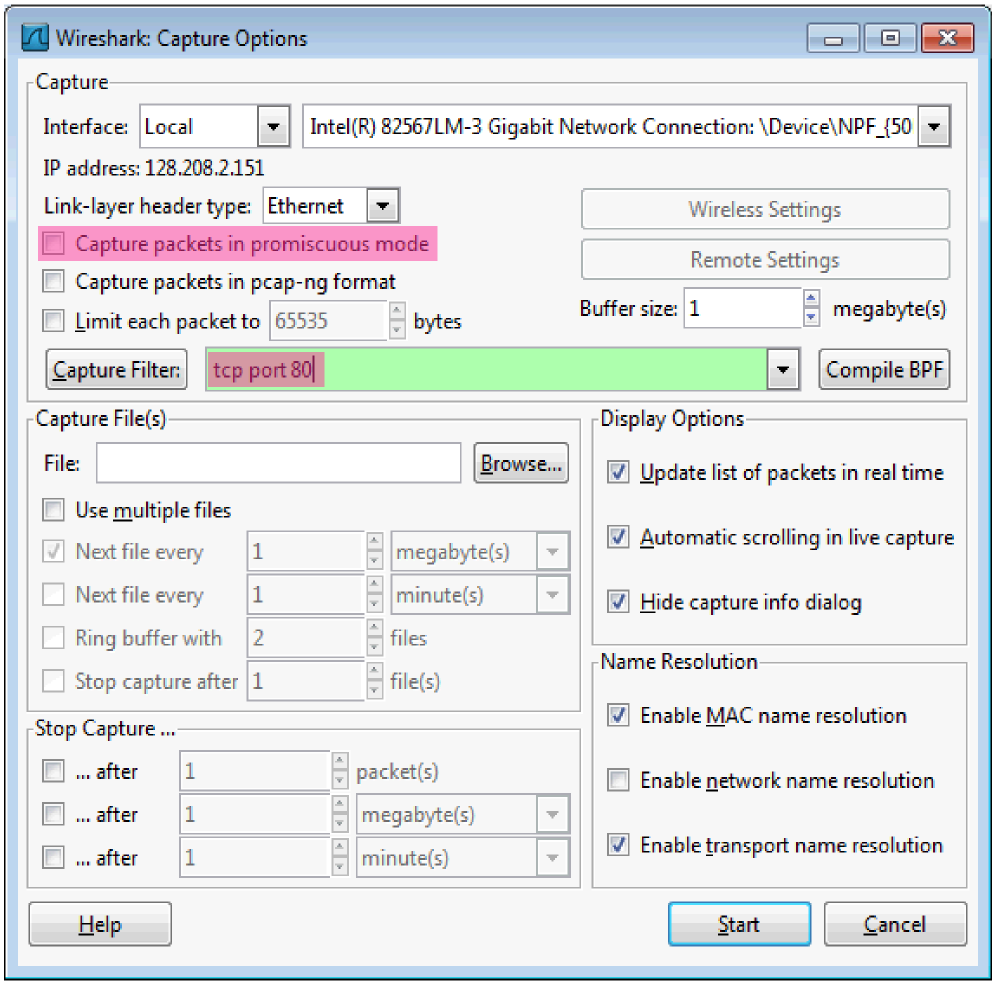
*Inspect your request and response to answer the following questions:*

1. *What version of HTTP is the server running?*
2. *How is the beginning of the content sent by the server recognized by the client?*
3. *How does the client know what type of content is returned?*

## Step 2: 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 (and other browsers!) to minimize HTTP traffic. When you clear your browser cache, do not delete your cookies if you have a choice.
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. Your capture window should be similar to the one pictured below, other than our highlighting. Select the interface from which to capture as the main wired or wireless interface used by your computer to connect to the Internet. If unsure, guess and revisit this step later if your capture is not successful. Uncheck "capture packets in promiscuous mode". This mode is useful to overhear packets sent to/from other computers on broadcast networks. We only want to record packets sent to/from your computer. Leave other options at their default values. The capture filter, if present, is used to prevent the capture of other traffic your computer may send or receive. On Wireshark 1.8, the capture filter box is present directly on the options screen, but on Wireshark 1.9, you set a capture filter by double-clicking on the interface.


Wireshark: Capture Options

Capture

Interface: Local Intel(R) 82567LM-3 Gigabit Network Connection: \Device\NPF_{50

IP address: 128.208.2.151

Link-layer header type: Ethernet

Capture packets in promiscuous mode

Capture packets in pcap-ng format

Limit each packet to 65535 bytes

Buffer size: 1 megabyte(s)

Capture Filter: tcp port 80

Capture File(s)

File:

Use multiple files

[✓ Next file every] 1 megabyte(s)

Next file every 1 minute(s)

Ring buffer with 2 files

Stop capture after 1 file(s)

Stop Capture ...

... after 1 packet(s)

... after 1 megabyte(s)

... after 1 minute(s)

Display Options

[✓ Update list of packets in real time]

[✓ Automatic scrolling in live capture]

[✓ Hide capture info dialog]

Name Resolution

[✓ Enable MAC name resolution]

Enable network name resolution

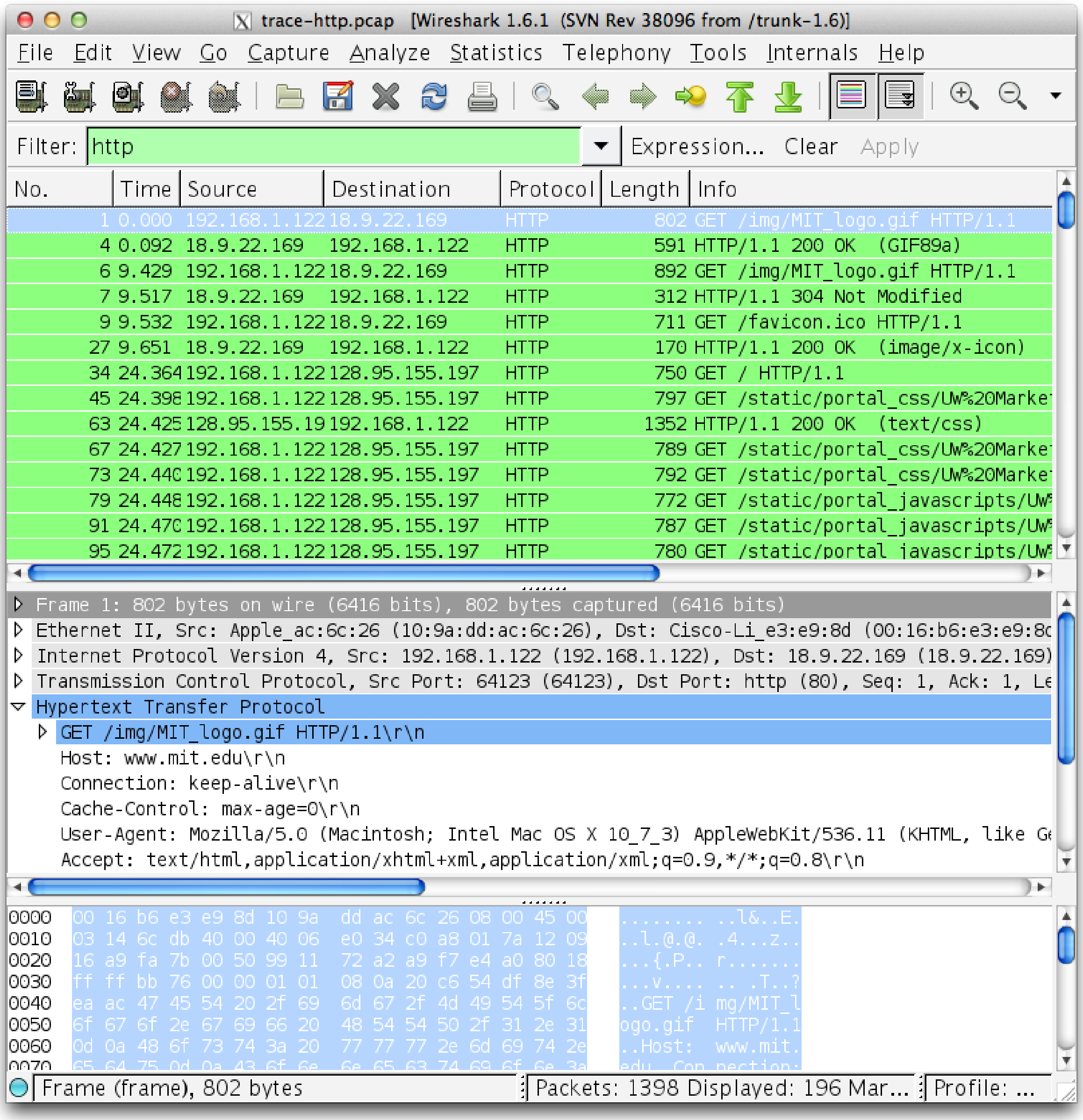
[✓ Enable transport name resolution]

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.


trace-http.pcap [Wireshark 1.6.1 (SVN Rev 38096 from /trunk-1.6)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter: http
Expression... Clear Apply

```
No.   Time    Source         Destination    Protocol Length Info
    1 0.000   192.168.1.122  18.9.22.169    HTTP      802 GET /img/MIT_logo.gif HTTP/1.1
    4 0.092   18.9.22.169    192.168.1.122  HTTP      591 HTTP/1.1 200 OK  (GIF89a)
    6 9.429   192.168.1.122  18.9.22.169    HTTP      892 GET /img/MIT_logo.gif HTTP/1.1
    7 9.517   18.9.22.169    192.168.1.122  HTTP      312 HTTP/1.1 304 Not Modified
    9 9.532   192.168.1.122  18.9.22.169    HTTP      711 GET /favicon.ico HTTP/1.1
   27 9.651   18.9.22.169    192.168.1.122  HTTP      170 HTTP/1.1 200 OK  (image/x-icon)
   34 24.364  192.168.1.122  128.95.155.197 HTTP      750 GET / HTTP/1.1
   45 24.398  192.168.1.122  128.95.155.197 HTTP      797 GET /static/portal_css/UW%20Marke
   63 24.425  128.95.155.19  192.168.1.122  HTTP     1352 HTTP/1.1 200 OK  (text/css)
   67 24.427  192.168.1.122  128.95.155.197 HTTP      789 GET /static/portal_css/UW%20Marke
   73 24.440  192.168.1.122  128.95.155.197 HTTP      792 GET /static/portal_css/UW%20Marke
   79 24.448  192.168.1.122  128.95.155.197 HTTP      772 GET /static/portal_javascripts/UW
   91 24.470  192.168.1.122  128.95.155.197 HTTP      787 GET /static/portal_javascripts/UW
   95 24.472  192.168.1.122  128.95.155.197 HTTP      780 GET /static/portal_javascripts/UW
```

```
▷ Frame 1: 802 bytes on wire (6416 bits), 802 bytes captured (6416 bits)
▷ Ethernet II, Src: Apple_ac:6c:26 (10:9a:dd:ac:6c:26), Dst: Cisco-Li_e3:e9:8d (00:16:b6:e3:e9:8d
▷ Internet Protocol Version 4, Src: 192.168.1.122 (192.168.1.122), Dst: 18.9.22.169 (18.9.22.169)
▷ Transmission Control Protocol, Src Port: 64123 (64123), Dst Port: http (80), Seq: 1, Ack: 1, Le
▽ Hypertext Transfer Protocol
  ▷ GET /img/MIT_logo.gif HTTP/1.1\r\n
    Host: www.mit.edu\r\n
    Connection: keep-alive\r\n
    Cache-Control: max-age=0\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/536.11 (KHTML, like Ge
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
```

```
0000  00 16 b6 e3 e9 8d 10 9a  dd ac 6c 26 08 00 45 00   ........ ..l&..E.
0010  03 14 6c db 40 00 40 06  e0 34 c0 a8 01 7a 12 09   ..l.@.@. .4...z..
0020  16 a9 fa 7b 00 50 99 11  72 a2 a9 f7 e4 a0 80 18   ...{.P.. r.......
0030  ff ff bb 76 00 00 01 01  08 0a 20 c6 54 df 8e 3f   ...v.... .. .T..?
0040  ea ac 47 45 54 20 2f 69  6d 67 2f 4d 49 54 5f 6c   ..GET /i mg/MIT_l
0050  6f 67 6f 2e 67 69 66 20  48 54 54 50 2f 31 2e 31   ogo.gif  HTTP/1.1
0060  0d 0a 48 6f 73 74 3a 20  77 77 77 2e 6d 69 74 2e   ..Host:  www.mit.
0070  65 64 75 0d 0a 43 6f 6e  6e 65 63 74 69 6f 6e 3a   edu..Con nection:
```

```
Frame (frame), 802 bytes    Packets: 1398 Displayed: 196 Mar...    Profile: ...
```

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

## Step 3: 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.

You are likely to see a variety of other headers too, depending on your browser, server, and choice of content that you requested.

*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?*

**Turn-in**: Answers to the above questions.

## Step 4: 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?*

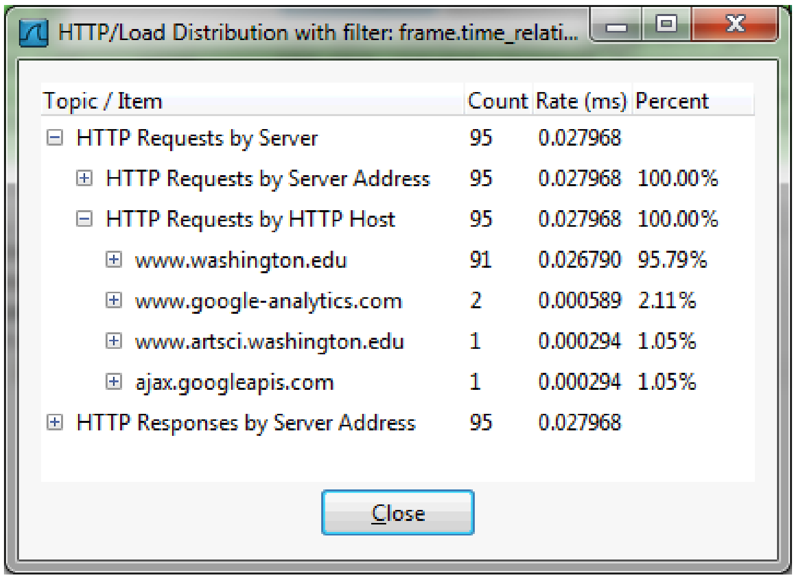
**Turn-in**: Answers to the above questions.

## Step 5: 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'll 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.

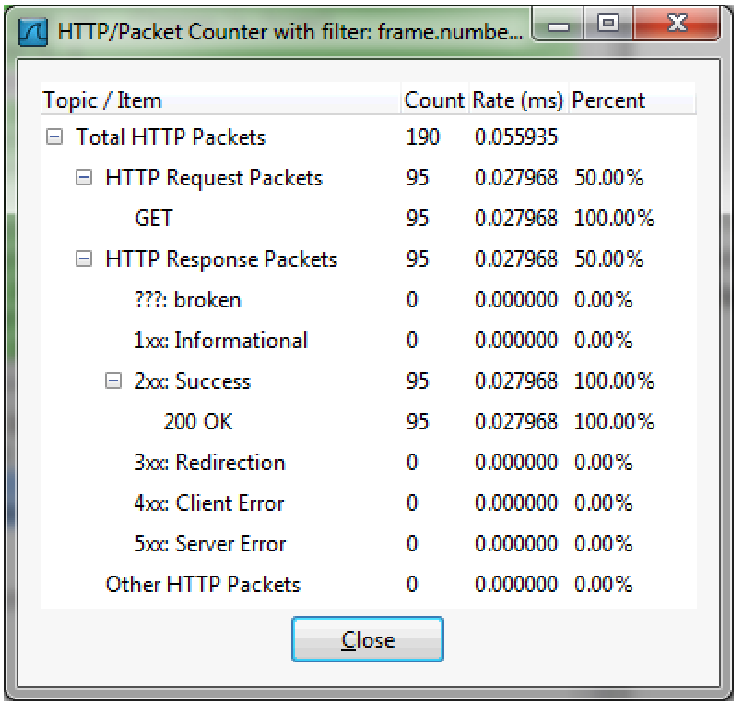
Looking at this panel will tell you how many requests were made to which servers. Chances are that your fetch will request content from other servers you might not have suspected to build the page. These other servers may include third parties such as content distribution networks, ad networks, and analytics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


HTTP/Load Distribution with filter: frame.time_relati...

| Topic / Item | Count | Rate (ms) | Percent |
|---|---|---|---|
| ⊟ HTTP Requests by Server | 95 | 0.027968 | |
| ⊞ HTTP Requests by Server Address | 95 | 0.027968 | 100.00% |
| ⊟ HTTP Requests by HTTP Host | 95 | 0.027968 | 100.00% |
| ⊞ www.washington.edu | 91 | 0.026790 | 95.79% |
| ⊞ www.google-analytics.com | 2 | 0.000589 | 2.11% |
| ⊞ www.artsci.washington.edu | 1 | 0.000294 | 1.05% |
| ⊞ ajax.googleapis.com | 1 | 0.000294 | 1.05% |
| ⊞ HTTP Responses by Server Address | 95 | 0.027968 | |

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.

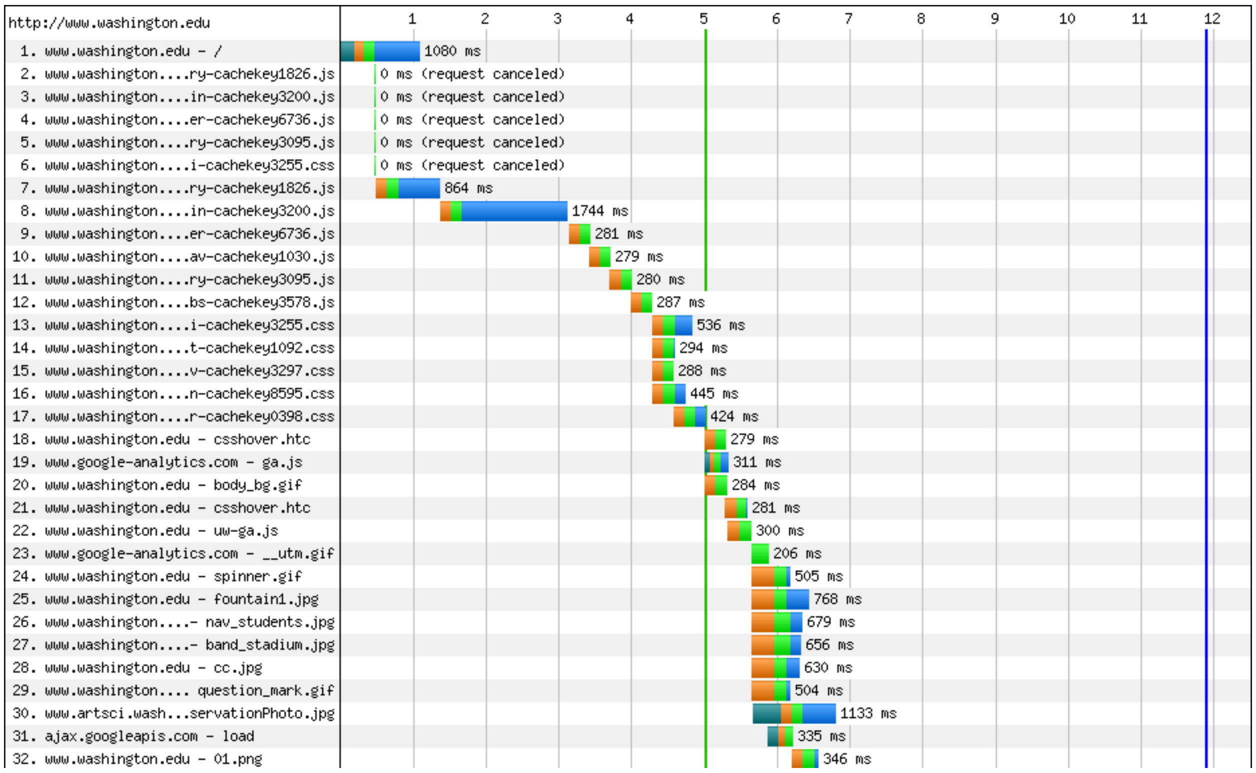

HTTP/Packet Counter with filter: frame.numbe...

| 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% |

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)

*There is no turn-in for this step.*

## Explore Your Network

We encourage you to explore HTTP on your own once you have finished this lab. Some suggestions:

- Look at how an HTTP POST works. We focused on the GET method above. POST is used to upload information to the server. You can study a POST by finding a simple web page with a form and tracing the form submission. However, do not study login forms as you want to observe an HTTP POST and not an encrypted HTTPS POST that is more typical when security is needed.

- 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 how HTTP GETs map to TCP connections once you have also done the TCP lab. With HTTP 1.1, the browser can make one TCP connection to a server and send multiple requests. Often after a single request the TCP connection will be kept open by the browser for a short while in case another request is coming. The number of concurrent connections and how long they are kept open depends on the browser, so you will discover how your browser behaves.
- 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.

[END]