Lab Simon Razniewski/Florian Klement

16. Wireshark II: DNS and HTTP

Objective

DNS (Domain Name System) is the system and protocol that translates domain names to IP addresses and more. HTTP (Hypertext Transfer Protocol) is used to transfer webpages.

Background: Network Setup

In a typical network, your computer contacts a local DNS nameserver to resolve domain names to IP addresses. The local nameserver may be another computer in your company network, a computer at your ISP, or your wireless AP. It exchanges a series of messages with remote DNS nameservers all over the Internet to perform the resolution. The setup is as shown in the figure below.

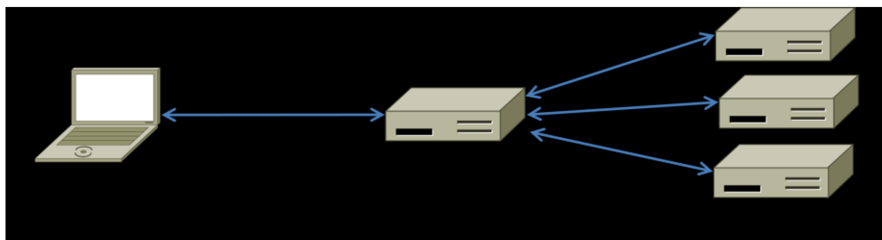

Figure 1: Typical network setup for DNS

It has an important implication: the trace we gather at our computer will see the exchanges between our computer and the local nameserver, but not between the local nameserver and the remote nameservers.

Task 1: Manual Name Resolution

Before we look at how your computer uses the DNS, we will see how a local nameserver resolves a DNS name, i.e., we will interact with remote nameservers. To do this exercise, you will pretend to be the local nameserver and issue requests to remote nameservers using the `dig` tool.

Pick a domain name to resolve, such as that of your web server. We use `magik-demo.inf.unibz.it/`. *Find the IP address of one of the root nameservers by searching the web.* For example, the Wikipedia article on root name servers includes the IP address of the root nameservers. Any one of these should do, as they hold replicated information. You need this information to begin the name resolution process, and nameservers are provided with it as part of their configuration.

Use `dig` *(for Windows you need to download it) to issue a request to a root nameserver to perform the first step of the resolution*. You are assuming that you have no cached information that will let you begin

a resolution below the root. The format of a `dig` command is "`dig @aa.bb.cc.dd domain-name`". It instructs `dig` to send a request to a nameserver at a given IP address (or name) for the given domain name. In the figure below, we used `dig` to send a request to the "`a`" root nameserver whose IP address is 198.41.0.4 to resolve our example web server, i.e., "`dig @198.41.0.4 www.uwa.edu.au`". The reply from the root does not provide the full name resolution, but it does tell us about nameservers closer to having the information for you to contact. In this case, it is nameservers who know about the ".it" domain. Multiple nameservers are given as alternative choices, and the reply helpfully includes their IP addresses; we can see IPv6 addresses as well as IPv4 addresses.

Continue the resolution process with `dig` *until you complete the resolution. When you have alternatives to choose, prefer IPv4 nameservers and select the first one in alphabetical order. If this nameserver has multiple IP addresses then select the numerically smallest IP address.*

Task: Draw a figure that shows the sequence of remote nameservers that you contacted and the domain for which they are responsible.

Task 2: Capture an HTTP Trace

Capture a trace of your browser making HTTP requests as follows:

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. 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.
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.
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. Uncheck "capture packets in promiscuous mode".
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*.

d. *Stop the capture after the fetches are complete.*

Task 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?***

Task 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?***

Task 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 re-

moving 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!

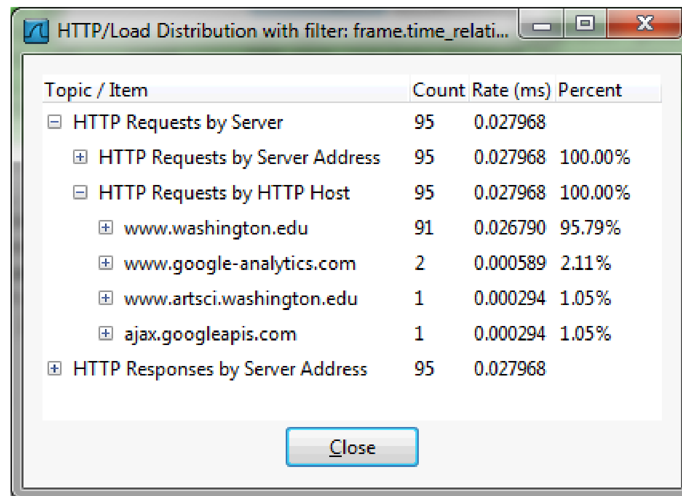

Figure 4: HTTP Load Distribution panel

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.

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. 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.

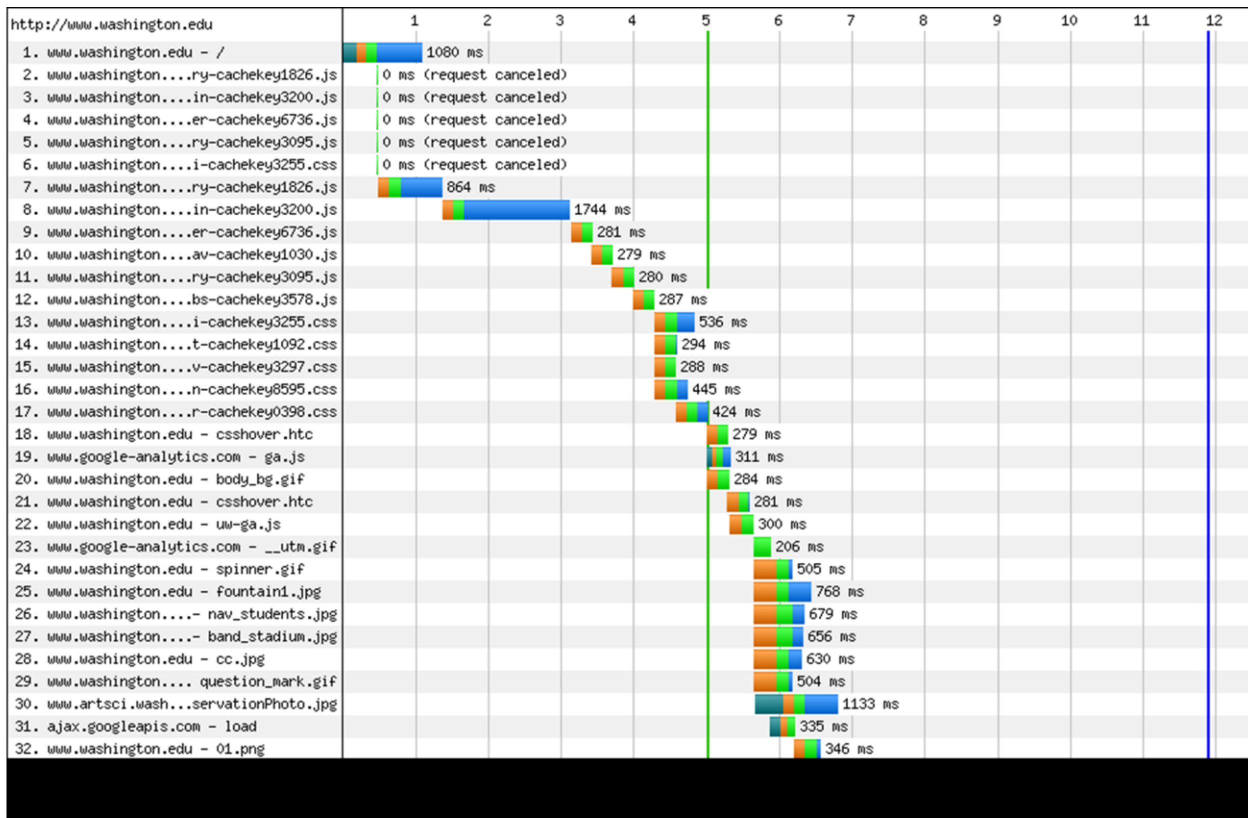

Figure 6: Start of waterfall graph for www.washington.edu (from pageloadtest.org)

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. 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.