Work in progress title

(A LaTeX class)

Anton Fagerberg
ada10afa@student.lu.se

January 15, 2015

Master's thesis work carried out at Jayway.

Supervisor: Roger Henriksson, Roger.Henriksson@cs.lth.se

Examiner: Görel Hedin, Gorel.Hedin@cs.lth.se

Abstract

This document describes the Master's Thesis format for the theses carried out at the Department of Computer Science, Lund University.

Your abstract should capture, in English, the whole thesis with focus on the problem and solution in 150 words. It should be placed on a separate right-hand page, with an additional *1cm* margin on both left and right. Avoid acronyms, footnotes, and references in the abstract if possible.

Leave a *2cm* vertical space after the abstract and provide a few keywords relevant for your report. Use five to six words, of which at most two should be from the title.

Keywords: MSc, template, report, style, structure

Acknowledgements

If you want to thank people, do it here, on a separate right-hand page. Both the U.S. *acknowledgments* and the British *acknowledgements* spellings are acceptable.

We would like to thank Lennart Andersson for his feedback on this template.

Contents

Chapter 1

Performance issues with HTTP requests

1.1 Headers

It is common in modern web applications to send a lot of HTTP requests. These requests can be very small, such as an GET request to update a field but the payload can also be considered very small when retrieving data from the server. Along with every HTTP request are plenty of headers. These headers can be a substantial part of every request and may therefore end up being the bottle neck if many small requests has to be performed.

As an example, consider the Instagram API[6] which has an end-point where you can get information about a certain user account serialised in JSON format. If a client was built which is supposed to show details about, for example, the 10 most popular users, we can benchmark how making 10 separate API requests would differ from concatenating them into one request through an API-gateway.

```
{
    "data": {
        "id": "1574083",
        "username": "snoopdogg",
        "full_name": "Snoop Dogg",
        "profile_picture": "http://distillery...",
        "bio": "This is my bio",
        "website": "http://snoopdogg.com",
        "counts": {
            "media": 1320,
            "follows": 420,
            "followed_by": 3410
        }
    }
}
```


Figure 1.1: User data response from the Instagram API.

Performing HTTP requests can be benchmarked using cURL. To make the requests authentic, we tell cURL to use the default headers provided by the browser Firefox. These headers includes among other things browsers User-Agent, media types which are acceptable responses and so on.

```
curl --trace-ascii - 'http://localhost:9000/user/snoopdogg'
    -H 'Host: localhost:9000' -H 'User-Agent: Mozilla/5.0 (
    Macintosh; Intel Mac OS X 10.10; rv:36.0) Gecko/20100101
     Firefox/36.0' -H 'Accept: text/html,application/xhtml+
    xml,application/xml;q=0.9,*/*;q=0.8' -H 'Accept-Language
    : en-US,en;q=0.5' --compressed -H 'Connection: keep-
    alive' -H 'Pragma: no-cache' -H 'Cache-Control: no-cache
    '
```

Figure 1.2: cURL command used in the benchmarks.

Performing this request will give us the following results:

```
=> Send header, 355 bytes (0x163)
0000: GET /user/snoopdogg HTTP/1.1
001e: Accept-Encoding: deflate, gzip
003e: Host: localhost:9000
0054: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.10; rv:36.
0094: 0) Gecko/20100101 Firefox/36.0
00b4: Accept: text/html,application/xhtml+xml,application/
   xml;q=0.9,*/
00f4: *;q=0.8
00fd: Accept-Language: en-US,en;q=0.5
011e: Connection: keep-alive
0136: Pragma: no-cache
0148: Cache-Control: no-cache
0161:
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 47 bytes (0x2f)
0000: Content-Type: application/json; charset=utf-8
<= Recv header, 21 bytes (0x15)
0000: Content-Length: 286
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 286 bytes (0x11e)
```

Figure 1.3: Results from cURL when performing a HTTP request to fetch one user. Actual payload omitted.

We can from this information see that 355 bytes are sent as header data, 87 bytes are then received as header data (17 + 47 + 21 + 2) and the actual payload is 286 bytes. This means 61% of every request to this end-point are nothing but header data.

If we instead would expose an endpoint where all 10 users could be requested with one HTTP request which instead would return an array of JSON objects, we'll get the following result:

```
=> Send header, 446 bytes (0x1be)
0000: GET /users/snoopdog1,snoopdog2,snoopdog3,snoopdog4,
   snoopdog5,sno
0040: opdog6,snoopdog7,snoopdog8,snoopdog9,snoopdo10 HTTP
   /1.1
0079: Accept-Encoding: deflate, gzip
0099: Host: localhost:9000
00af: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.10; rv:36.
00ef: 0) Gecko/20100101 Firefox/36.0
010f: Accept: text/html,application/xhtml+xml,application/
   xml;q=0.9,*/
014f: *;q=0.8
0158: Accept-Language: en-US,en;q=0.5
0179: Connection: keep-alive
0191: Pragma: no-cache
01a3: Cache-Control: no-cache
01bc:
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 47 bytes (0x2f)
0000: Content-Type: application/json; charset=utf-8
<= Recv header, 22 bytes (0x16)
0000: Content-Length: 2871
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 2871 bytes (0xb37)
```

Figure 1.4: Results from cURL when performing a HTTP request to fetch 10 users. Actual payload omitted.

From the results, we can see that the sent headers are increased from 355 to 446 bytes because of the longer URL which specifies all users to fetch. The received headers are increased with just one byte from 87 to 88 because of the increased *Content-Length* field. This results in a total header size of 534 bytes. The payload is increased from 286 bytes to 2871 bytes - about 10 fold as expected since we request 10 users at once instead of one at a time, the small increase is because of the array-characters in the JSON format. By avoiding doing 10 separate requests and instead do one concatenated request, the overhead from all headers have now been reduced from 61% to 16% and this will scale according to the number of requests - the more requests concatenated, the lesser amount of overhead from HTTP headers.

#	10 users, 10 request	10 users, 1 request
Headers	4,420 B	534 B
Payload	2,860 B	2,871 B
% headers of total data	61%	16%

Figure 1.5: Header and payload ratio when doing 10 separate or one concatenated request to fetch users.

The data displayed in this example should be viewed as a lower bound. In practise, HTTP cookies which are used for personalisation, analytics and session management are also sent with every HTTP request an can add up to multiple kilobytes of protocol overhead for every HTTP request[4, p. 200].

This is one of the issues that may be mitigated by using HTTPS/2.0 which knows which headers that has already been sent and therefore doesn't need to retransmit them on subsequent requests[4, p. 222].

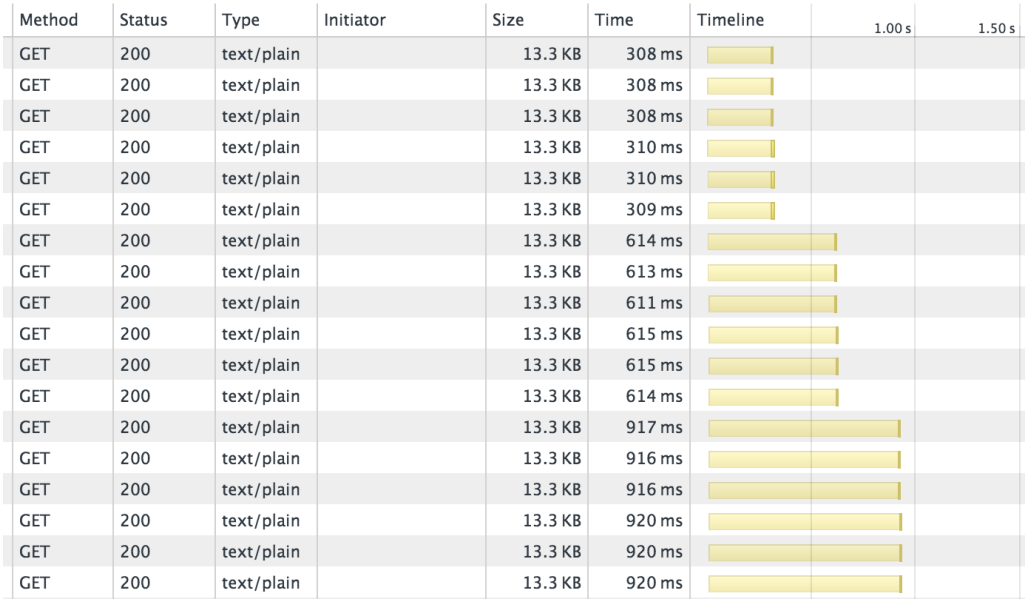
1.2 Maximum TCP connections

The HTTP 1.X protocol doesn't allow data to be multiplexed over the same connection[4, p.194]. For this reason, browser vendors has introduced a connection pool of 6 TCP connections per host (the HTTP 1.1 specification limits it to 2 connections[2], but modern browsers has refused to conform to this).

A common way to deal with this issue is domain sharding. Since the limit of 6 TCP connections are on a host basis, it is possible to create multiple subdomains to mitigate this problem. If the subdomains {shard1, shard2, ...}.example.com was created and pointed to the same server, more than 6 TCP connections can be used in parallel at the same time from that machine. This approach is not without downsides as every new host requires a new DNS lookup, a TCP 3-Way handshake and a TCP slow start from TCP which can slow down the user experience[4, p. 199] - just the DNS lookup typically takes 20-120 ms[8, p. 63]. - just the DNS The browser always opens 6 connections per shard automatically even if not all of them are used. Sharding can be a complicated manual process and it is hard to decide how many shards to use for optimal performance.

As an example, we can look at downloading thumbnails for an image gallery. Suppose we want to download 60 base64-encoded thumbnails and the connection we are using has a lot of bandwidth but high latency.

If all images were downloaded by using a single HTTP request per thumbnail, we can clearly see that the 6 TCP connection limit will become the bottleneck.



Method	Status	Type	Initiator	Size	Time	Timeline 1.00 s 1.50 s
GET	200	text/plain		13.3 KB	308 ms	
GET	200	text/plain		13.3 KB	308 ms	
GET	200	text/plain		13.3 KB	308 ms	
GET	200	text/plain		13.3 KB	310 ms	
GET	200	text/plain		13.3 KB	310 ms	
GET	200	text/plain		13.3 KB	309 ms	
GET	200	text/plain		13.3 KB	614 ms	
GET	200	text/plain		13.3 KB	613 ms	
GET	200	text/plain		13.3 KB	611 ms	
GET	200	text/plain		13.3 KB	615 ms	
GET	200	text/plain		13.3 KB	615 ms	
GET	200	text/plain		13.3 KB	614 ms	
GET	200	text/plain		13.3 KB	917 ms	
GET	200	text/plain		13.3 KB	916 ms	
GET	200	text/plain		13.3 KB	916 ms	
GET	200	text/plain		13.3 KB	920 ms	
GET	200	text/plain		13.3 KB	920 ms	
GET	200	text/plain		13.3 KB	920 ms	

Figure 1.6: Chrome developer tools showing how the 6 TCP connection limit becomes a bottle neck on a connection with 300 ms of latency.

We can calculate the total delay caused by latency in our example by this formula:

$$\text{total latency} = \text{number of thumbnails} * \frac{\text{latency per request}}{\text{nr of parallel requests}} \tag{1.1}$$

In our example, we fetch 60 thumbnails and with a latency of 300 ms per request. Our browser can handle 6 parallel connections which gives us the following result:

$$\text{total latency} = 60 * \frac{300}{6} = 3,000 \text{ ms} \tag{1.2}$$

If we instead could concatenate these 60 thumbnail requests into one request to an API-gateway which in turn would fetch the 60 thumbnails and respond with all of them at once - then we would only have to pay the latency cost once which would reduce the total latency from 3,000 ms to 300 ms.

It is worth pointing out that increasing bandwidth would not resolve this problem as the latency is the only bottleneck in this scenario. It is not uncommon for browsers to wait 100 - 150 ms before spending 5 ms to download an image which means that latency is accounting for 90-95% of the total time for the HTTP request[5].

1.2.1 Chunked responses

When fetching thumbnails, you often want to display them as soon as each individual image is loaded and not wait for all of them to load before displaying them. When using a concatenated request to fetch all of them at the same time, the server can use chunked transfer encoding in the HTTP response to send the thumbnail data in chunks[1]. By doing so, images can be loaded as soon as they are available, even out of order. This

technique has been implemented by Dropbox as their solution for improved performance when displaying thumbnails[7].

1.3 Compression

An API-gateway can compress the requested data before it is sent to the client. A common compression algorithm is Gzip (GNU Zip) which works best on text-based files, such as HTML, CSS and JavaScript. Gzip has an expected compression rate of 60-80% when used on text-based files[4, p. 237].

It is worth mentioning that there are scenarios where Gzip compression applied to very small files can increase the total size because of the Gzip dictionary overhead[3].

As an example, user data for 50 users was created and stored in JSON format. When this was requested from the server without compression, the total content-length of the HTTP request payload was 55,205 bytes. When tunnelling this request through the API-gateway which compresses this data using Gzip, the content length was reduced to 16,563, which amounts to a space saving of 70%.

$$\text{Space Saving} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}} = 1 - \frac{16,563}{55,205} = 70\% \tag{1.3}$$

An important thing to note about Gzip compression is that only the payload is compressed in HTTP 1.1[9]. This means that the headers including cookies are not compressed which would've otherwise been an additional performance gain. This is one of the improvements which have been addressed in HTTP/2[4, p. 222].

Bibliography

[1] J Fielding, R. Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing - chunked transfer coding. http://tools.ietf.org/html/rfc7230#section-4.1.

[2] J. Mogul J. Frystyk H. Masinter L. Leach P. Berners-Lee T Fielding, R. Gettys. Hypertext transfer protocol – http/1.1. http://www.ietf.org/rfc/rfc2616.txt.

[3] I Grigorik. Optimizing encoding and transfer size of text-based assets. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer.

[4] I Grigorik. *High-Performance Browser Networking*. O'Rilley Media, Inc., 2013.

[5] B Hoffman. Bandwidth, latency, and the size of your pipe. http://zoompf.com/blog/2011/12/i-dont-care-how-big-yours-is.

[6] instagram.com/developer. User endpoints. http://instagram.com/developer/endpoints/users/#get_users.

[7] Z Mahkovec. Improving dropbox performance: Retrieving thumbnails. https://tech.dropbox.com/2014/01/retrieving-thumbnails/.

[8] S Souders. *High Performance Web Sites*. O'Rilley Media, Inc., 2007.

[9] D Stenberg. Http transfer compression. http://daniel.haxx.se/blog/2011/04/18/http-transfer-compression/.