Critical Resources and the First 14 KB - A Review

Category: Blog

This page was originally created on 05-Aug-2019 and last edited on 28-Jun-2022.

Introduction

Many web performance experts recommend putting all critical resources in the first 14 KB of your web page. This is based on a bit of understanding of TCP which underlies each HTTP connection – at least for now until HTTP/3 and QUIC come along. But is it really true? In my book HTTP/2 in Action I suggested that this 14 KB statistic was no longer really that relevant in the modern world, if it ever was, and was asked to expand upon this - hence this post. But let me caveat it by saying that web performance is massively important, and there are many, many, many use cases showing this, so I'm not arguing against it and people should optimise this. Just don't get too hung up on this 14 KB number.

The basics of TCP and where this 14 KB number comes from

TCP is a guaranteed delivery protocol and uses a number of methods to achieve this. First it acknowledges all TCP packets and resends any unacknowledged packets. Additionally it acts pretty nice to the network and starts slow and builds up to full capacity in a process known as *TCP slow start*, gently feeling its way and checking it's not overwhelming anything and leading to lost (unacknowledged) packets. The combination of these two things means that when TCP starts it allows 10 TCP packets to be sent, before they must be acknowledged. As those packets are acknowledged it allows more packets to be sent, doubling up to allow 20 packets to be sent next time, then 40, then 80...etc. as it gradually builds up to the full capacity the network can handle. The 14 KB magic number is because each TCP packet can be up to 1500 bytes, but 40 of those bytes are for TCP to use (TCP headers and the like) leaving 1460 bytes for actual data. 10 of those packets means you can deliver 14,600 bytes or about 14 KB (14.25 KB actually).

Now networks are incredibly fast – pretty close to the speed of light actually – and it usually only takes 10s or 100s of milliseconds for those responses to travel back and forth between server and client. However, one of the few things in the known universe faster than the speed of light, is users impatience. So delaying sending more data, while you wait for those acknowledgments, means you are waiting for at least one back and forth between client and server (known as a *round trip*) and this introduces an unwanted performance bottleneck. It would be much better if the first batch of sends could contain all the critical data to start the browser on its way to drawing (aka *rendering*) the web page.

Another point to note is that HTML is actually read by most browsers as a stream of bytes and so you don't need to download the whole of HTML before the browser starts to process it. Basically browsers are pretty impatient too – because of those impatient users – and so will start looking at HTML as soon as it starts arriving. It may see references to CSS, JavaScript and other resources and start fetching those so it can render the page as quickly as possible. Therefore, even if you can't send the whole page in the first 14 KB (though please do if you can!), having as much critical data in that first 14 KB will allow the browser to start working on that data earlier.

While 14 KB may not seem a lot in these days of multi-megabyte pages, remember that you are not trying to fit your entire page into that 14 KB limit (though again do if you can!), but only trying to optimise what the browser sees in that first chunk of data. Delivering all your critical resources in the first 14 KB therefore gives you the best chance of maximising the browser's first read and should lead to a faster page, hence why web performance experts have been giving that advice. In an ideal world, that will even be enough to start rendering if you inline critical CSS – which I don't actually like btw! – but even if you can't get it as far as a one round-trip render, getting the browser to start downloading all the required resources as quickly as possible will also help.

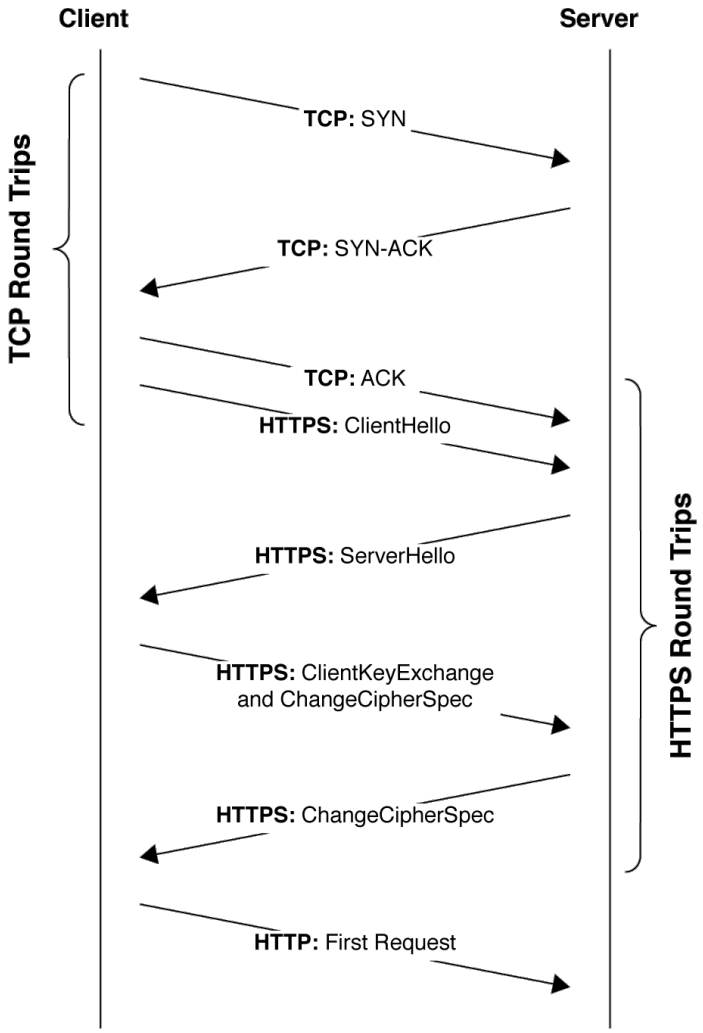
14 KB and the modern web

However that advice may not be entirely accurate and personally I think fixating on that magic 14 KB number isn't actual that helpful. It makes several assumptions, that aren't really realistic on the web of today, if they ever were.

First up is the assumption that all of that 14 KB will be used to deliver the HTML. Even in the old world of plaintext HTTP that wasn't the case with HTTP response codes (`200 OK`) and HTTP response headers taking up some of that. HTTP response headers in particular can be massive! Twitter's Content Security Policy header is approximately 5 KB alone for example. Yes HTTP/2 allows HTTP response headers to be compressed, but that

basically works by storing headers from the previous requests and referring to them on subsequent requests. This means the first request - when we are most concerned about this 14 KB limit - pretty much uses the full sized headers – though that's not 100% accurate as HTTP/2 can still use an initial static table for common headers and use Huffman encoding rather than ASCII for slightly smaller headers, but that is only a partial optimisation and the larger benefit is from reuse for second and subsequent requests. Since I've just introduced HTTP/2 that brings the second problem here - HTTPS and HTTP/2. Both of these require some additional messages to be exchanged to establish the connection.

HTTPS requires two round trips to set up the TLS connection, assuming the most commonly used TLSv1.2 and no session resumption here, as shown in the following handshake diagram:



So that's at least 2 of your 10 TCP packets for sending used up at least. TLSv1.3 allows 1 round trip (1-RTT) – or even 0 round trips (0-RTT) in certain scenarios – which is one of the big benefits of it. However I don't think that changes the argument I'm about to make too much, because additionally TLS certificates can be quite large - easily requiring multiple TCP packets to be sent. So using up 2 TCP packets is probably your **best** case even for TLSv1.3 and certainly for TLSv1.2.

Then let's assume you're using HTTP/2 as many top sites are using now - especially those that care about performance. HTTP/2 is only available over HTTPS (for browsers at least) and requires a few more messages to be exchanged to set up the HTTP/2 connection. For a start the HTTP/2 connection preface message must be sent by the client first, then a SETTINGS Frame MUST be sent by each side, and often one or more WINDOWS_UPDATE frames are also sent at the beginning of the connection. Only after all this can the client send the HTTP request. Now it's true that those HTTP/2 frames do not need to be separate TCP packets and also do not need to be acknowledge before client requests can be made, but they do eat into this initial 10 TCP packet initial limit. Additionally, though it's hardly worth mentioning since it is so small, but each HTTP/2 frame is also preceded by a header of at least 9 bytes, depending on the exact frame type, which further easts into that 14.25 KB limit – if we're gonna count the TCP packet overhead then only seems fair to do the same for HTTP/2 packet overhead!

So if this 10 packet / 14 KB stat was accurate, then we'd be down to at least half of that (2 packets for TLS handshake, 2 for the HTTP/2 connection set up responses and 1 for HTTP Header response leaving 5 packets), which sounds much worse! However, on the plus side, some of those back and forth messages will have resulted in TCP ACKs meaning we will have **more** than 14 KB when we come to sending back the HTML not **less**. For

example TLS requires the clients to respond during the handshake, which, as we shall soon see, means they can also acknowledge some of those previously sent TCP packets at the same time, increasing the congestion window size, so we will already have increased beyond that 10 packet limit.

There is also the assumption that only 10 TCP packets can be sent. While that is true of most modern operating systems, it is a relatively new change, and previously 1, 2 and then 4 packets were used as this limit. It's safe to assume, unless you are running in legacy hardware (and let's assume those worried enough about performance to be looking at this 14 KB limit are not), then that it is at least 14 KB, however some CDNs have even started going higher than 10. So far I have not seen any proposals to suggest increasing this in general for servers, but again those interested in performance may well be using a CDN.

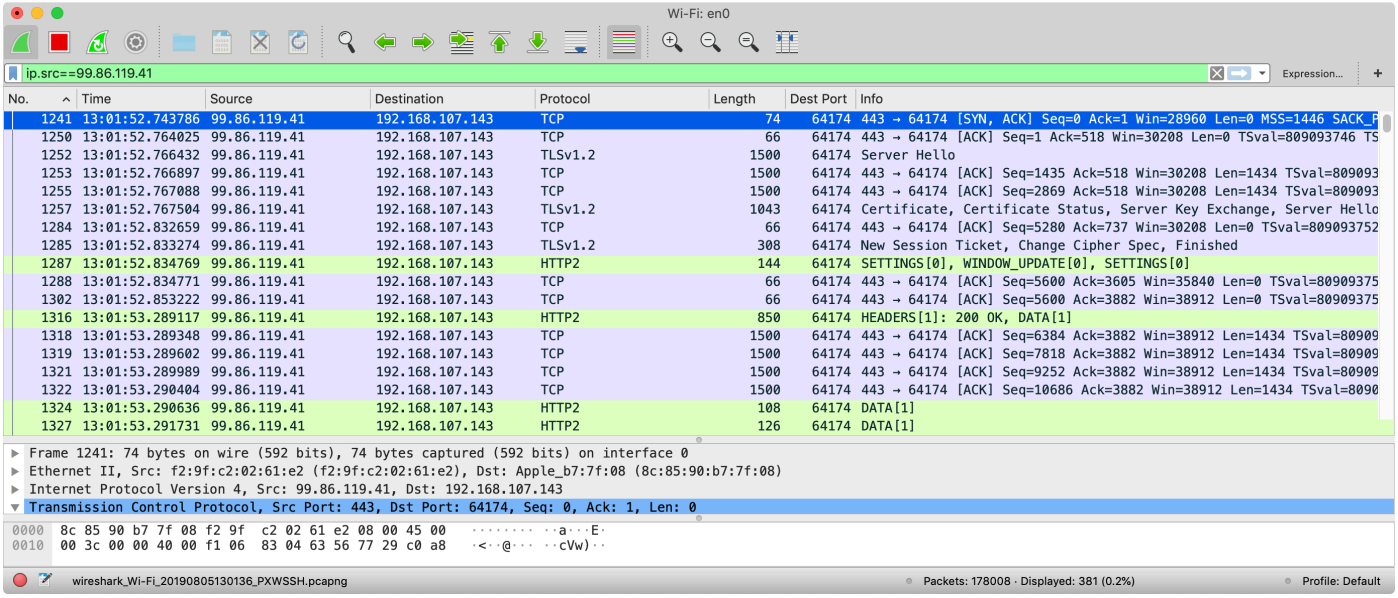
Additionally, as alluded to earlier, this 14 KB number has always been based on a somewhat flawed assumption - that TCP usage is such an exact and clean protocol. It's fine to say that TCP stacks can send up to 10 packets without acknowledgement, and that each acknowledgement of all the packets in flight will double the congestion window size meaning after this initial 10 packets, 20 packets can be sent, then 40 packets, 80 packets...etc. However life is rarely that clinical. The reality is that TCP will be acknowledging packets all the time depending on the TCP stack and the timings involved. The 10 packet limit is just a **maximum** unacknowledged limit, but quite often an acknowledgement may be sent after, say, 5 packets. Or even after 1. This is especially true when the client has to send data anyway (such as part of the TLS handshake, or as part of setting up the HTTP/2 connection), so the reality is that often (always?) the congestion window will be larger than 10 KB by the time you come to send the HTML anyway.

Finally it should also be remembered that HTML should be delivered compressed (gzipped or using the newer brotli compression) so 10 KB could easily be 50 KB once that is taken into consideration, as text (HTML, CSS and JavaScript) compresses very well.

Real Life Example

Since one of my points above, is that the 14 KB is based on theory, rather than actual usage, let's look at a real life example. What follows is just one example rather than a definitive study or exhaustive analysis of the top X sites, but at least it will show whether I'm talking complete rubbish or if there is anything in these ramblings. The findings can easily be repeated and if someone wishes to, they can repeated this at a larger scale. Anyway, for this example I fired up Chrome with SSL Key Logging to allow Wireshark to intercept all requests. How this works, for those readers unfamiliar with this, is beyond the scope of this blog post, but there are others that will show you the way. This set up basically means that Chrome is working as normal but allowing Wireshark to sniff on all traffic back and forth, even HTTPS encrypted traffic, so we can see exactly what is going on at a TCP packet level.

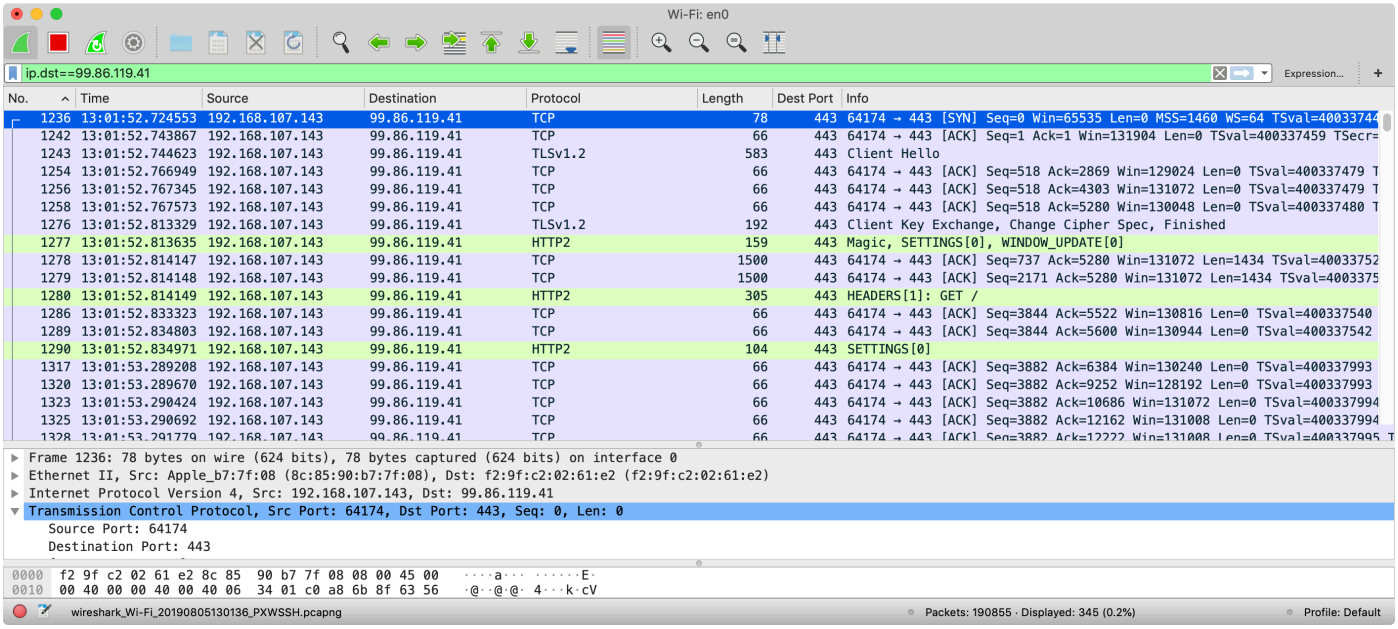
Next I connect to a well-known website (I've chosen https://www.amazon.com) and, after checking the IP address that I connected to in Chrome's developer tools, I filtered the Wireshark capture to just that traffic – as shown in the screenshot below:



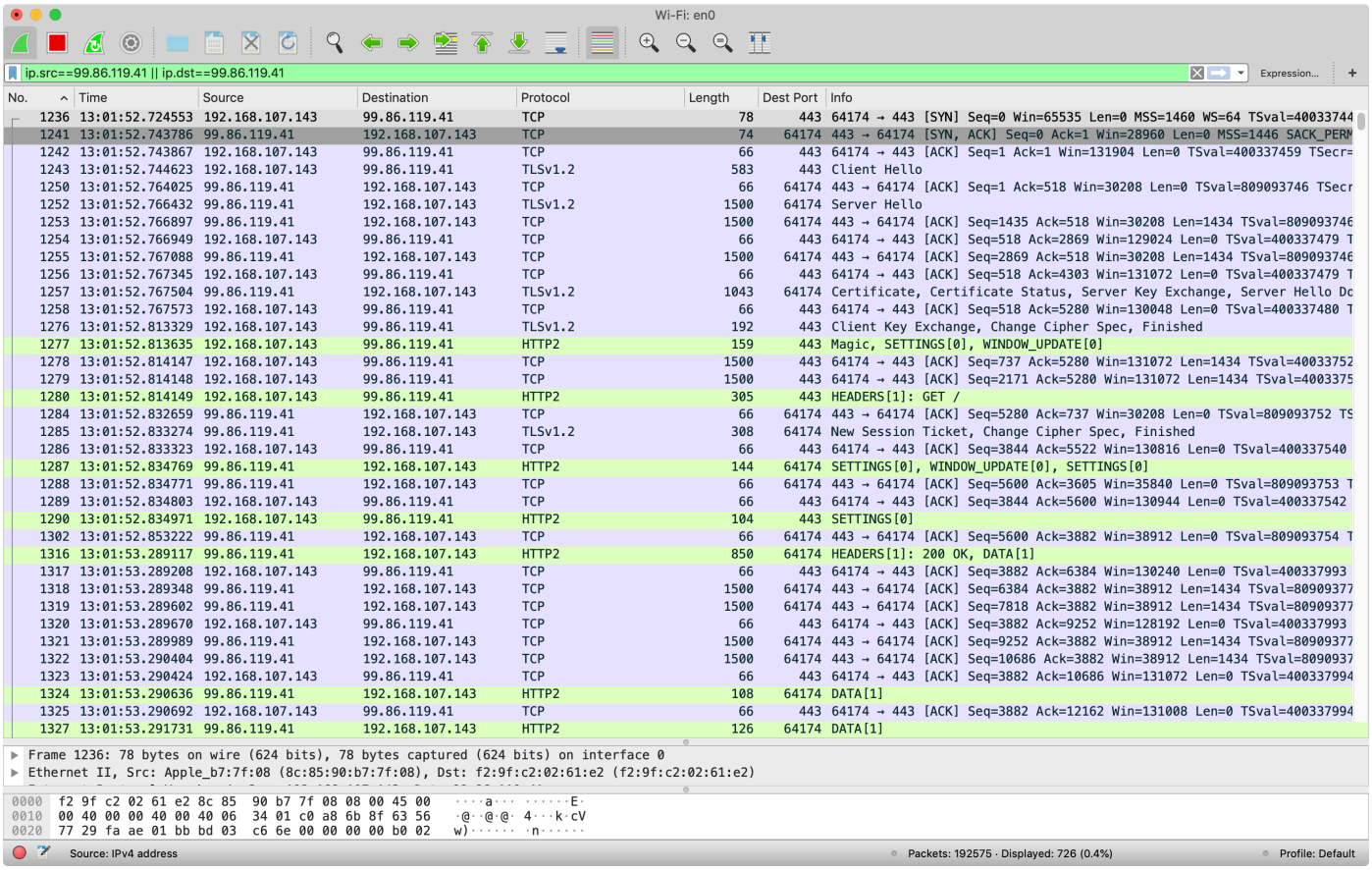
For those not used to Wireshark, this can be a little confusing, but basically each line is a message being sent on the network. Where possible Wireshark will classify the message type at the highest protocol level (e.g. TCP, TLS or HTTP/2) for that message. It will also fall back to TCP for message fragments (e.g. for TLS messages that span multiple TCP messages so cannot be recognised as a TLS message until the final TCP packet).

In the above screenshot you can see all the messages coming back from Amazon and first of all we see message 1241 which is the server side of the TCP handshake (SYN, ACK), then a TCP ACK in message 1250, then we see the TLSv1.2 `Server Hello` message (1252), followed by 3 messages (1253, 1255 and 1257) needed to send the next part of the TLS handshake with the Certificate and the Key Exchange data (note only the last of these is marked as TLSv1.2 as the first two are just fragments so are marked as TCP). Also in the first two of these you can see the maximum 1500 byte size I discussed earlier. TCP packets are really only this maximum size when there is a stream of data to send like this. After that there's another random TCP ACK (1284) before we finish out the TCP handshake (1285). Next up we're into the HTTP/2 setup messages described above - lucky these are small so can fit in single TCP packet. After this we have a couple more TCP ACKs before we get to the first real HTTP message (1316) where we send the response headers (including the `200` status code shown in the screenshot). After this we have a few more ACKs before we start to send the HTML which takes 5 packets (1318, 1319, 1321, 1322 and 1324) for the first HTTP/2 DATA frame (note I have not shown all the DATA frames in this screenshot for brevity sake). Now the congestion window size is not something that's actually transmitted, so we can only guess what the size is by this point but I think it's clear to show that a lot more than just the HTML body is being sent back, so the assumption that all those 10 packets are just for the HTML is certainly not true.

Let's have a look at the client side now though by changing the filter at the top to look at `ip.dst` instead:

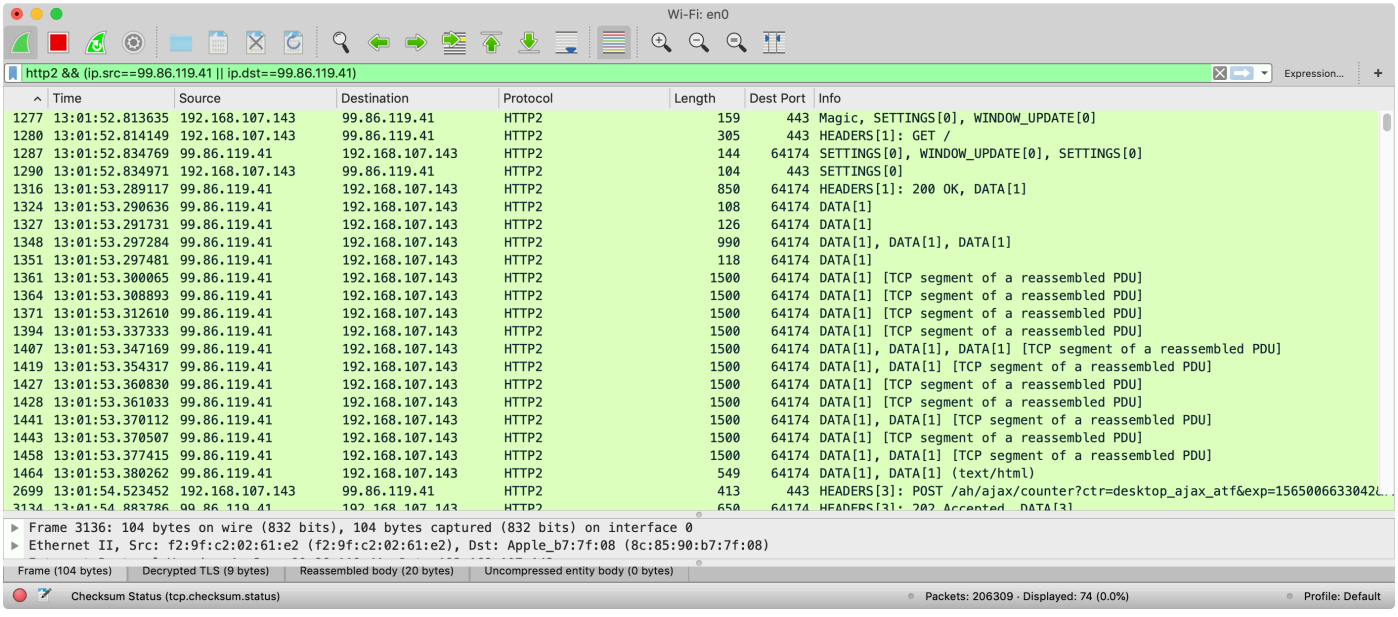


Here we see a similar story, with the TCP set up, TLS messages, HTTP/2 messages all before we even request the home page. Each of those TCP packets of 66 bytes are just TCP ACKs and nothing else, so this shows that TCP ACKs are happening all the time - even if there is no other traffic being sent. However when there is other traffic being sent (e.g. TLS messages or HTTP/2 messages) there are **still** TCP ACKs going on as part of those messages too. In all 14 messages, all with TCP ACKs, are sent to the server after the initial TCP handshake before the initial `GET` request is even sent (and this request also includes an ACK so we're up to 15 packets). And likely some more ACKs are sent before the HTML is started to be delivered, and even more while it's being delivered. In fact let's look at both sides at once (which can initially be confusing, hence why I showed each side first):



Here we can see more ACKs were sent from client to server in packets 1286, 1289, 1317, 1320, 1323 before the first DATA frame was returned by the server, meaning we have ACKed 20 packets at least, as the HTML starts flowing, so we're somewhere between 28 KB and 57 KB that can be sent unacknowledged. Additionally, as I stated above, more are ACKed as the HTML data flows as can be seen in message 1325 and more messages off screen. It's difficult to say if the HTML was ever blocked waiting for a TCP acknowledgement but I suspect not. The Amazon home page is not small (113 KB gzipped, or 502 KB uncompressed), but due to the exponential increase of TCP slow start, we only need 3 or 4 round trips of acknowledgements to potentially get to the point where this can be sent in one go as we've already shown HTTPS and HTTP/2 connections will have a few round trips.

Looking at just the HTTP/2 messages we can see a steady flow of DATA frames with no noticeable delay between them which would be indicative or waiting for TCP acknowledgments, which seems to back up my suspicions that there was no delay:



I will be honest here though and with a 20 ms round trip time (that I measured through `ping`), it would be difficult to see such a delay anyway. A worse connection may show a bigger impact. Also it should be noted that using

Chrome's Dev Tools network throttling won't help here as that is artificial throttle within Chrome, rather than anything at the network layer. I did repeat the test with a 300 ms round trip website, and saw the same thing, but it's still very anecdotal since I am testing single sites here rather than carry out extensive field research.

The point remains however, that there are lots of TCP messages back and forth so that 10 TCP / 14 KB number has been shown to be theoretical rather than something you would see in the real world - especially on an HTTPS connection (and even more so on an HTTP/2 connection over HTTPS).

Summary

Hopefully this post has explained why I think the 14 KB number is not an absolute number or something web developers should be adhering to, too rigidly ("Oh no - I've gone to 15 KB, must shave off another 1 KB or we can't launch!"). I'm sure that many of the web performance enthusiasts that have promoted this number did not intend for people to take it so literally, but the problem with a hard number like this, especially when backed up with a little bit of science, is that people will take it as such, when they should not.

This post is also not intended to mean that page size is not important - it is massively important and I'm a big fan of optimising web performance! However I do warn against absolutism. The real world is far more complex than that. The general advice still stands: put your important resources near the top of the page so the browser can see them as soon as possible and start working on them, and ideally make your page as small as possible to allow it to download quickly. But don't sweat too much over this magic 14 KB number.

Do you agree or disagree? Do you have any more data on this to confirm or refute what I've talked about above? Let me know below your thoughts below. And if this post interests you then check out my book - HTTP/2 in Action. The discount code `39pollard` will even get you 39% off if buying direct from Manning.

Want to read more?

More Resources on this subject

- HTTP/2 in Action - my book which delves into HTTP and TCP a lot more.
- High Performance Browser Networking by Ilya Grigorik which offers a tonne of low level network type facts and information.
- Awesome talk by Patrick Meenan on how browsers load and render resources, particularly in an HTTP/2 world.

Related Posts on TuneTheWeb.com

- Other performance posts.
- Why do we need HTTP/2? - an excerpt from "HTTP/2 in Action".
- HTTP/2.
- HTTP/2 Server Push.
- Inlining CSS is Not for Me.
- Implementing Accelerated Mobile Pages.
- Further Performance Resources.

This page was originally created on 05-Aug-2019 and last edited on 28-Jun-2022.

How useful was this page?