CS 168 Summer 2026

DNS, HTTP, Ethernet

Discussion 8

1 True/False

1.1 Pipelined connections are frequently used in practice.

False. Pipelined connections are not commonly used because of (a) bugs and (b) head-of-line blocking. HTTP2 uses multiplexing which avoids these shortcomings.

1.2 Hosts usually perform the iterative DNS resolution process themselves.

False; hosts use local DNS servers to perform DNS lookup.

1.3 Every zone always has at least 2 name servers.

True.

1.4 When looking up a root server, BGP will use unicast to find the correct root server.

False; BGP uses anycast to find the closest root server.

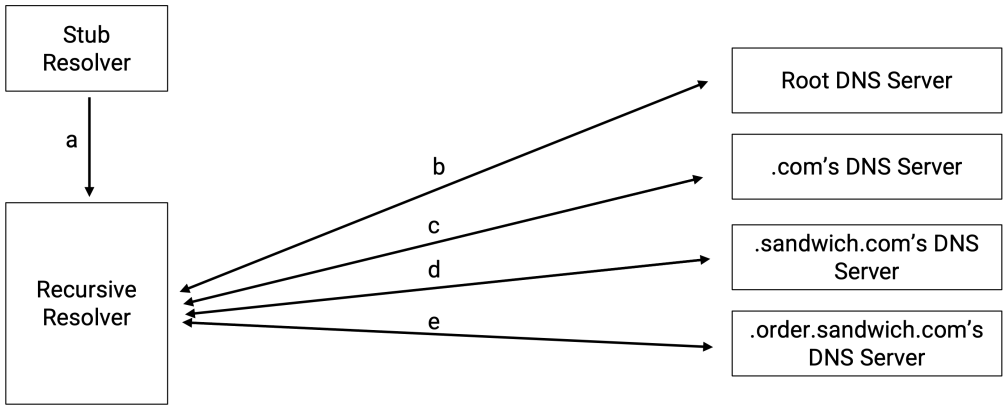
1.5 A client can establish a TCP connection with a root server.

False; TCP requires keeping the state, but anycast does not guarantee all the packets of the same connection are sent to the same root server.

1.6 Most queries to DNS root servers are for nonexistent TLDs.

True.

2 Domain Name System



A sandwich ordering website `www.order.sandwich.com` is accepting online orders for the next T minutes. Consider the following setup of DNS servers, with annotated latencies between servers.

Assume that:

- The latency between your computer and the website's server is t.
- Once you send an order for a sandwich, you must wait for a confirmation response from the website before issuing another.
- Your computer **does not cache** the website's IP address.

Your job is to calculate the total number of sandwiches that can be ordered in time T in each condition.

2.1 Your local DNS server doesn't cache any information.

Your computer begins by issuing a DNS query for `www.order.sandwich.com` to its local DNS server, which takes time a. Your local DNS server then **iteratively queries** the root DNS server, `.com`'s DNS server, `.sandwich.com`'s DNS server, and `order.sandwich.com`'s DNS server, which takes time $2b + 2c + 2d + 2e$. It then returns the result of the query to you, which takes time a.

After obtaining the IP address, your computer **issues a sandwich request** to `www.order.sandwich.com`, which responds with an order confirmation, taking time $2t$.

The **total time per sandwich order** is: $\mathbf{2a + 2b + 2c + 2d + 2e + 2t}$

Thus, the **total number of sandwiches** that can be ordered in time T is: $\left\lfloor \frac{T}{2a+2b+2c+2d+2e+2t} \right\rfloor$

2.2 Your local DNS server caches responses, with a time-to-live $L \geq T$.

If $L \geq T$, once the result of the DNS query for `www.order.sandwich.com` is cached, it remains cached in our local DNS server until the website's server ultimately goes down.

- The **first query** takes the same amount of time as before: $2a + 2b + 2c + 2d + 2e + 2t$
- **Subsequent queries** take only $\mathbf{2a + 2t}$, since the local DNS server can now provide the IP immediately.

Thus, the **total number of sandwiches** we can order is:

$$\begin{cases} 1 + \left\lfloor \frac{T-(2a+2b+2c+2d+2e+2t)}{2a+2t} \right\rfloor & \text{if } T \geq 2a + 2b + 2c + 2d + 2e + 2t \\ 0 \text{ otherwise} \end{cases}$$

2.3 Let $T = 600$ seconds and $a = b = c = d = e = t = 1$ second. Your local DNS server caches responses with a finite TTL of **30 seconds**.

When the first DNS query is made, the response gets cached in the **local DNS server** at:

$a + 2b + 2c + 2d + 2e = 9$ seconds

This response remains cached **for 30 seconds**, expiring at **time 39 seconds**.

- The **first order is completed at time**: 12 seconds $(9s + a + 2t)$
- **Additional orders** can be placed from time **12 to 39**, each taking **4 seconds** $(2a + 2t)$.
- The number of orders made during this cached period is: $\left\lceil \frac{39-12}{4} \right\rceil = 7$
- The **pattern repeats every 40 seconds** ($T = 600$ allows for **15 cycles**): $\frac{600}{40} = 15$
- Total sandwiches ordered: 15 × (7 + 1) = **120**

Not bad!

3 Performance

We want to download a webpage. We must first download the HTML (size P). This HTML includes URLs for two embedded images of size M which need to then be loaded. Assume the following:

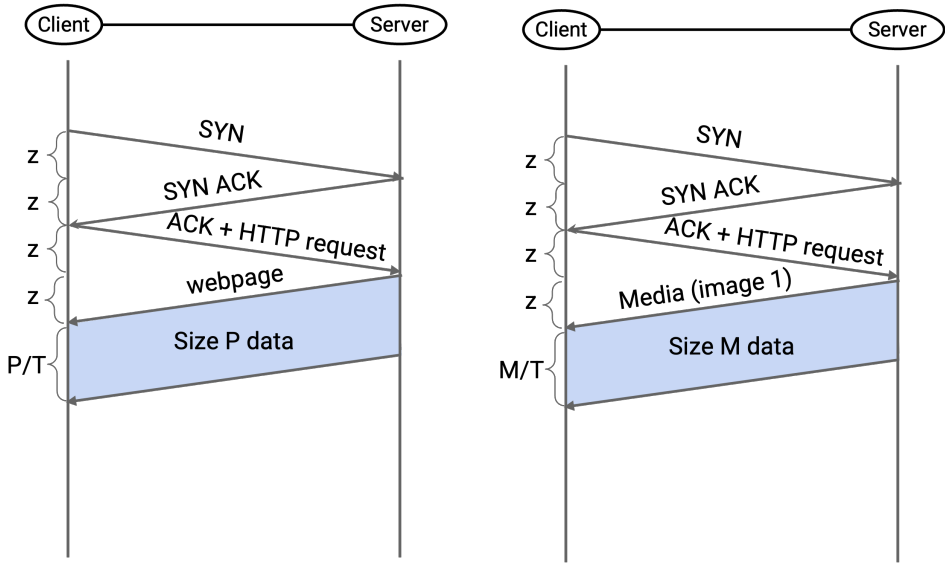
- SYN, ACK, SYNACK, and HTTP request packets are small and take time z to reach their destination in either direction.
- Each of our HTTP connections can achieve throughput T for sending files and web pages across the network unless there are concurrent connections, in which case each connection's throughput is divided by the number of concurrent connections.
- You never need to wait for TCP connections to terminate.

For each of the following scenarios, compute the total time to download the web page and both media files.

3.1 Sequential requests with non-persistent TCP connections.

(a) $2z$ (SYN + SYNACK) $+z$ (ACK/HTTP request) $+\left(\frac{P}{T}+z\right)$ (actual webpage)
(b) $2z$ (SYN + SYNACK) $+z$ (ACK/HTTP request) $+\left(\frac{M}{T}+z\right)$ (first media file)
(c) $2z$ (SYN + SYNACK) $+z$ (ACK/HTTP request) $+\left(\frac{M}{T}+z\right)$ (second media file)

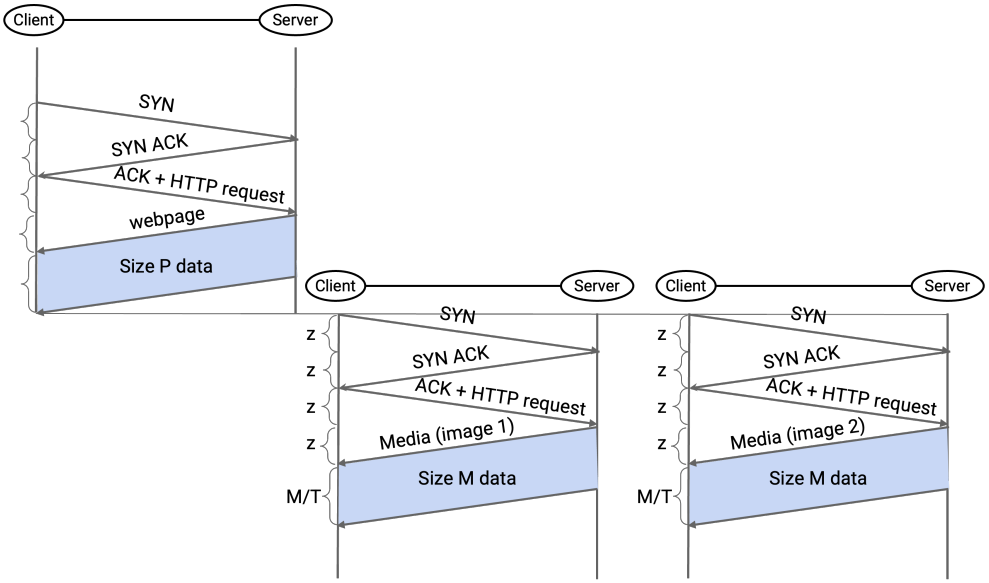
Total $= 12z + \frac{P}{T} + 2\frac{M}{T}$



3.2 Concurrent requests with non-persistent TCP connections.

(a) $2z$ (SYN + SYNACK) $+z$ (ACK/HTTP request) $+\left(\frac{P}{T}+z\right)$ (actual webpage)
(b) $2z$ (SYNs + SYNACKs) $+z$ (ACKs/HTTP requests) $+\left(\frac{M}{\left(\frac{T}{2}\right)}+z\right)$

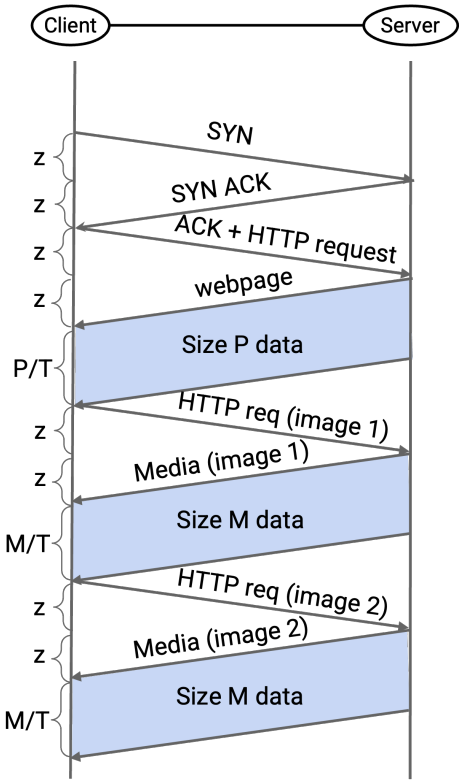
Total $= 8z + \frac{P}{T} + 2\frac{M}{T}$



3.3 Sequential requests with a single persistent TCP connection.

(a) $2z$ (SYN + SYNACK) $+z$ (ACK / HTTP request) $+\left(\frac{P}{T}+z\right)$ (actual webpage)
(b) z (HTTP request) $+\left(\frac{M}{T}+z\right)$ (first media file)
(c) z (HTTP request) $+\left(\frac{M}{T}+z\right)$ (second media file)

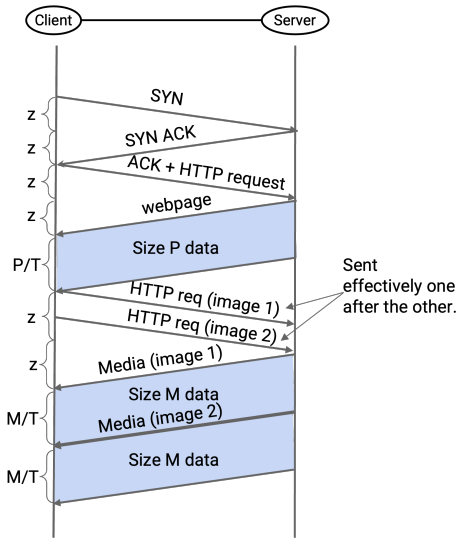
Total $= 8z + \frac{P}{T} + 2\frac{M}{T}$



3.4 Pipelined requests within a single persistent TCP connection.

(a) $2z$ (SYN + SYNACK) $+z$ (ACK / HTTP request) $+\left(\frac{P}{T}+z\right)$ (actual webpage)
(b) z (Both HTTP requests)
(c) $\frac{M}{T}$ (first media file)
(d) $\left(\frac{M}{T}+z\right)$ (second media file)

Total $= 6z + \frac{P}{T} + 2\frac{M}{T}$ Note: The second media file can begin to be sent immediately after the first is pushed onto the wire; hence, we don't need to account for the propagation delay of the first file in our calculation.

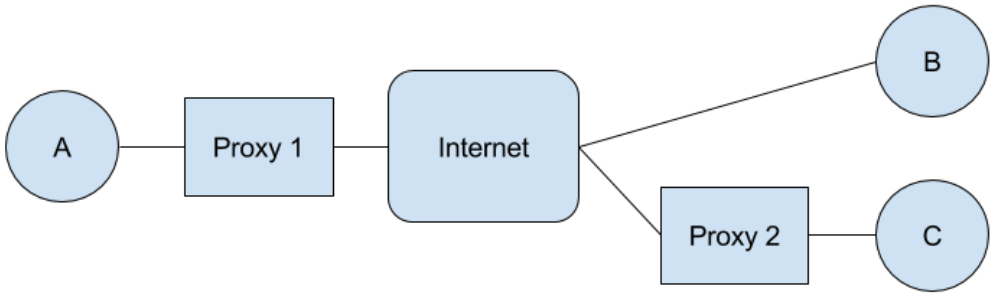


3.5 We have been assuming that the throughput for sending media files is T for a single connection, and $\frac{T}{n}$ for n concurrent connections. Remember that the throughput for sending the media files depends on both its transmission delay and propagation delay. So far we ignored this finer granularity division but depending on the size of the media files, we can make more inferences about how fast we can send the media files. If the media files are very small, what kind of delay would dominate the time it would take to send them? What if the files are very large?

If the media files are very small, then transmission delay is small, so propagation delay dominates. If the media files are very large, then transmission delay is large, and so dominates.

Takeaway: Adding concurrency, re-using existing TCP connections, and pipelining all help to speed up end-to-end page loads.

4 HTTP



Consider the (abstracted) network topology above. Hosts A and C are connected to HTTP proxies that cache the results of the last two HTTP requests they've seen to improve performance. The proxy will perform any TCP handshakes or teardowns with the client and server concurrently. That is, when it gets a SYN from a client, it will respond and immediately send a SYN to the server, and similarly for other messages.

As an example, let's suppose that A sends a request to B. A sends a SYN to B, which is intercepted by the proxy. The proxy sees the request is going to B, and initiates its own TCP handshake with B, all the while completing the original, separate handshake with A. By the time this has completed, there will be 2 TCP sessions: the first between A and the proxy, and the second between the proxy and B. When A sends a request, the proxy will forward it to B if it is not cached, or respond if it is cached. A similar process to the handshake is followed when A tears the connection down.

For the purposes of this problem, assume that the latency on each link is L, and the latency through the internet is I. Processing delay at all points and packet size is assumed to be negligible (don't consider transmission and processing delay). Assume that TCP connections use the 3-message teardown, and that no data is sent in ACK packets.

Suppose that Host A issues the following list of requests (in order):

- berkeley.edu to Host C
- eecs.berkeley.edu to Host C
- stanford.edu to Host B
- mit.edu to Host B
- stanford.edu to Host B
- berkeley.edu to Host C

4.1 What is the total time to complete all requests if they are issued one at a time? That is, each completes before the next is started with a separate TCP session (no need to wait for the session to be torn down).

To set up a connection to B, A will initiate a TCP connection with P1, which will establish a connection to B. The total time to establish the connection is the time for the SYN to reach P1 plus the time it takes P1 to establish a connection to B: $L + 2(2L + I)$

Since the proxy will send the request from A right after sending the ACK. **Note:** The SYNACK from P1 to A and SYN from P1 to B occurs concurrently, so we could write the term $2(2L + I)$ in the equation $\max(L,$ above more precisely as $2(2L + I))$, but that is equivalent to $2(2L + I)$.

To set up a connection to C, A will initiate a TCP connection with P1, which will do so with P2 (who it thinks is C), which will then do so with C. Similarly to before, the time this takes is: $L + 2(2L + I)$

Since the longest delay is between the two proxies, that's all we need to consider. Since this time is the same for both servers, we'll call it S, the setup time.

In all cases, A will complete its handshake with its proxy before the proxy finishes its handshake with the server. Therefore, the HTTP request from Host A will have already arrived at P1 and will be sent immediately after the proxy P1 finishes its handshake. This means that in the total request time for requests to B and C, the latency from A to P1 doesn't matter. This total request time is only affected by the latency from P1 to hosts B and C.

The latency from P1 to B is: $2L + I$ We'll call this L_B.

The latency from P1 to C is: $3L + I$ We'll call this L_C.

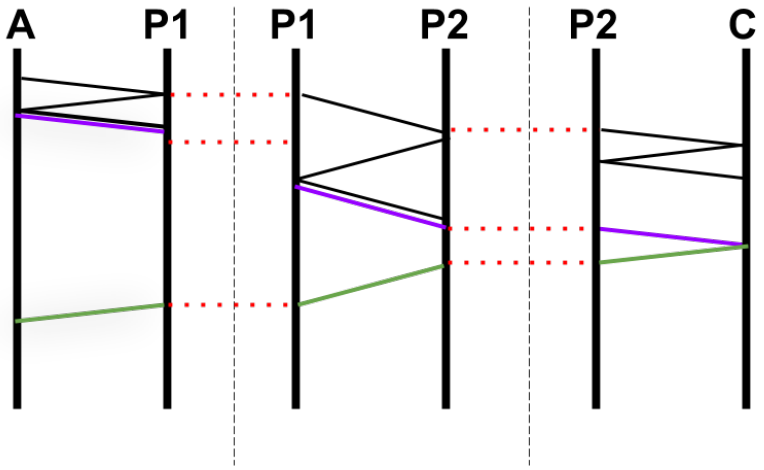
Even though they share the same subdomain base, the first request (berkeley.edu) and the second request (eecs.berkeley.edu) go to different servers (since eecs.berkeley.edu is hosted in a different place than berkeley.edu)—so they won't be helped by the cache. Therefore, they take: $S + 2L_C + L$ time each.

The third and fourth requests are similar, taking: $S + 2L_B + L$ time each.

The fifth request hits P1's cache, only taking: $2L + 2L$ time.

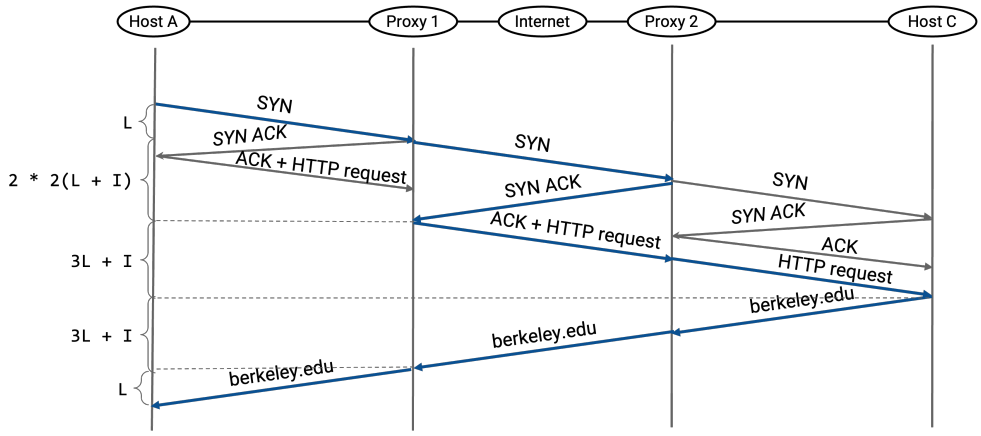
The sixth request hits P2's cache, taking: $S + 2(L_C - L) + L$ time.

Therefore, the total time is: $2(S + 2L_C + L) + 2(S + 2L_B + L) + 4L + S + 2(L_C - L) + L = 2S + 4L_C + 2L + 2S + 4L_B + 2L + 4L + S + 2L_C - 2L + L = 5S + 6L_C + 4L_B + 7L = 5(5L + 2I) + 6(3L + I) + 4(2L + I) + 7L = 25L + 10I + 18L + 6I + 8L + 4I + 7L = 58L + 20I$



Summer 2025 Interpretation: We'll follow the timestamps for each individual request that is made.

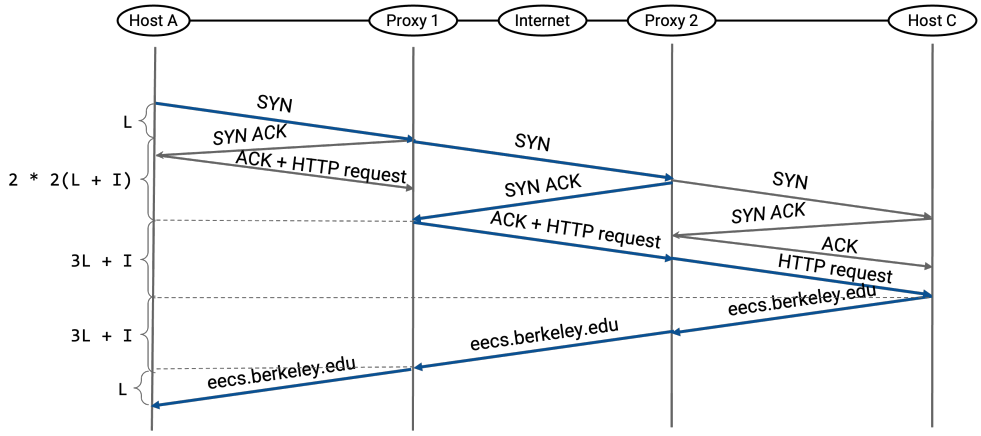
Request 1: *berkeley.edu to Host C.* Based on the diagram below, this takes a total of $12L + 4I$ time. We'll also need to add berkeley.edu to Proxy 1 and Proxy 2's cache.



Both proxies now have cached the entry for berkeley.edu.

Proxy	Entry 1	Entry 2
Proxy 1	berkeley.edu	-
Proxy 2	berkeley.edu	-

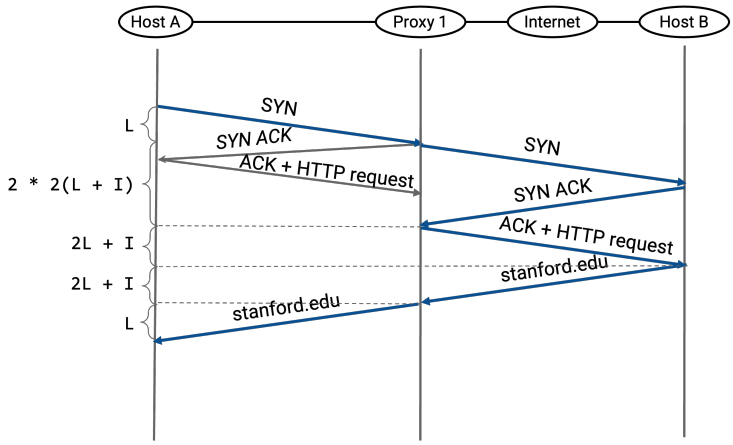
Request 2: *eecs.berkeley.edu to Host C.* Although eecs.berkeley.edu is a subdomain of berkeley.edu, its server is in a different location, and hence, its value isn't cached. Therefore, we have to make a full retrieval here, which mimics that of Request 1, for a total of $12L + 4I$ time. We'll also need to add eecs.berkeley.edu to Proxy 1 and 2's cache.



Both proxies now have cached the entry for eecs.berkeley.edu.

Proxy	Entry 1	Entry 2
Proxy 1	berkeley.edu	eecs.berkeley.edu
Proxy 2	berkeley.edu	eecs.berkeley.edu

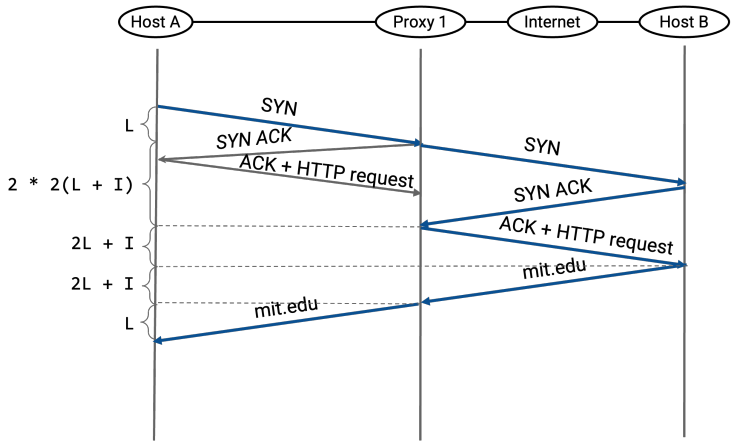
Request 3: *stanford.edu to Host B.* Based on the diagram below, this takes a total of $10L + 4I$ time. We'll also need to add stanford.edu to Proxy 1's cache, but not Proxy 2's, since this connection didn't go through Proxy 2.



Proxy 1 now has cached the entry for stanford.edu. In the process, since each cache can only have 2 elements, it has evicted berkeley.edu. Since the cache is FIFO, we've also moved eecs.berkeley.edu up.

Proxy	Entry 1	Entry 2
Proxy 1	~~berkeley.edu~~eecs.berkeley.edu	~~eecs.berkeley.edu~~stanford.edu
Proxy 2	berkeley.edu	eecs.berkeley.edu

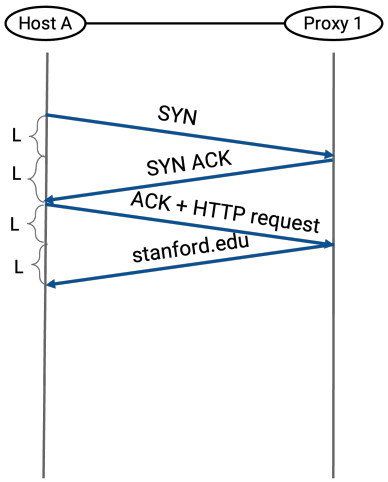
Request 4: *mit.edu to Host B.* Based on the diagram below, this takes a total of $10L + 4I$ time, as its pathing is identical to that of Request 3. We'll also need to add mit.edu to Proxy 1's cache (same as Request 3).



Proxy 1 now has cached the entry for mit.edu. In the process, since each cache can only have 2 elements, it has evicted eecs.berkeley.edu.

Proxy	Entry 1	Entry 2
Proxy 1	~~eecs.berkeley.edu~~stanford.edu	~~stanford.edu~~mit.edu
Proxy 2	berkeley.edu	eecs.berkeley.edu

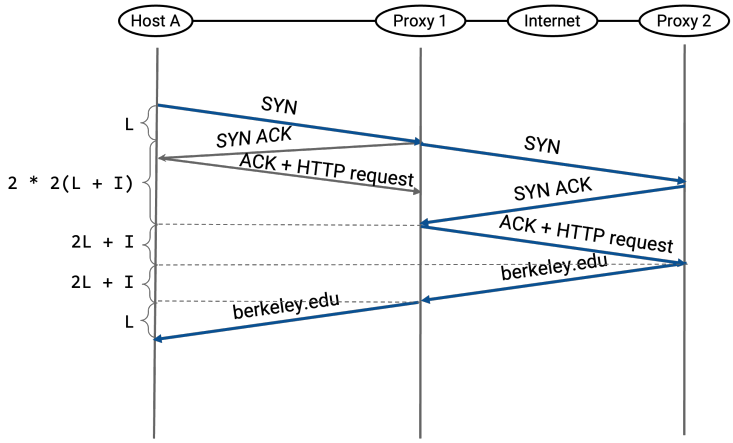
Request 5: *stanford.edu to Host B.* Based on the diagram below, this takes a total of $4L$ time. Note that our Proxy 1 cache currently contains the result of the request for stanford.edu, and therefore, we don't need to go any further! We'll also shift stanford.edu back to the most recent use of Proxy 1's cache.



We've moved Proxy 1's entries around to demonstrate that stanford.edu was the most recent request.

Proxy	Entry 1	Entry 2
Proxy 1	~~stanford.edu~~mit.edu	~~mit.edu~~stanford.edu
Proxy 2	berkeley.edu	eecs.berkeley.edu

Request 6: *berkeley.edu to Host C.* Based on the diagram below, this takes a total of $10L + 4I$ time. Note that Proxy 1 had long evicted berkeley.edu's data from its cache, so we have to go to Proxy 2 to retrieve.



Proxy 1 now has cached the entry for berkleey.edu, evicting mit.edu. Likewise, we've moved around the entries in Proxy 2 to reflect that berkeley.edu was the most recently accessed item.

Proxy	Entry 1	Entry 2
Proxy 1	~~mit.edu~~stanford.edu	~~stanford.edu~~berkeley.edu
Proxy 2	~~berkeley.edu~~eecs.berkeley.edu	~~eecs.berkeley.edu~~berkeley.edu

We can now sum up the time for all six requests to get the answer of $58L + 20I$.

4.2 What is the total time to complete all requests if they are performed concurrently such that everything is done in parallel and there is no possibility for caching?

Since all the requests occur concurrently, there is no opportunity for caching by the proxies. Therefore, the total time is just the **max** of all the times for the individual requests, which is the first request. It takes: $S + 2L_C + L = 5L + 2I + 2(3L + I) + L = 5L + 2I + 6L + 2I + L = 12L + 4I$