CSCI-1680
HTTP II

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, Scott Shenker and John Jannotti
Administrivia

• TCP is due next Tuesday

Will announce some final project info, grading feedback soon
Warmup

Browser wants to fetch: http://example.com/page.html

Assuming no caching, what is the minimum number of packets the browser needs to wait for?
It gets worse

Modern web traffic almost always uses HTTPS: `https://example.com/page.html` => Creates a secure transport layer to prevent eavesdropping, etc (more on this later)

Extra setup steps!
How does a browser load a page?

• Click a link, type in URL => browser fetches main page
• Main page has links to more resources => need to fetch these too!
  – Images, CSS, Javascript, etc.
How does a browser load a page?

• Click a link, type in URL => browser fetches main page
• Main page has links to more resources => need to fetch these too!
  – Images, CSS, Javascript, etc.

• New resources might load yet more resources…

Recursive process with many dependencies!
Early websites: not many dependencies, usually served by one server.
Now???
On a modern webpage…

• Huge number of dependencies, external resources
  – … from many different locations, not just one server!

• Lots of asynchronous operations => loading new resources as you are using the page

• Lots of dynamic content => generated by the server specifically for you (your feed, ad data, …)

How to make this fast?
How to make this fast?

What’s important for performance?
Observation: lots of small requests

Latency is a problem! Need many RTTs just to fetch one resource!

HTTP/1.0: One TCP connection per request!
Can we do better? (1997)
HTTP/1.1: Persistent connections
=> Reuse TCP connection to for multiple requests

**Problems**
- Multiple servers.
- Caching.
- Requests in order.

- NOT TRULY STATABLE ANYMORE.
- BUT NO HANDSHAKE EVERY TIME.

**Problems?**
Can we do better?

HTTP/1.1: Persistent connections
=> Reuse TCP connection to for multiple requests
Can we do better?

HTTP/1.1 (1996): Persistent connections

=> Reuse TCP connection to for multiple requests

Problems?

⇒ One big request blocks others => head of line blocking
⇒ Same if connection has packet loss
⇒ Doesn’t help when fetching from multiple locations
What can be done?

**Goal: Pipelining**

- **Multiple Connections** - Browsers have a pool of request threads (multiple servers)

- Would like to have multiple streams on same TCP connection.
HTTP/1.1 Request

GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh ...
Accept: text/xml,application/xml ...
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
What can be done?

Pipelining: have multiple “in-flight” requests at once

Two methods

• Multiple TCP connections in parallel

• Change the HTTP protocol: multiple requests per connection
What can be done?

Pipelining: have multiple “in-flight” requests at once

Two methods
• Multiple TCP connections in parallel
  => Browsers often do this (up to a limit)
• Change the HTTP protocol: multiple requests per connection
  => Newer HTTP versions: HTTP/2, HTTP/3
HTTP/2 (2015)

Adds support for multiplexed streams on one connection

TCP doesn’t know about multiple streams
=> If packet loss on one stream, others are blocked until packet comes in
=> Head of line blocking

What happens if a packet gets dropped?

https://www.twilio.com/blog/2017/10/http2-issues.html
HTTP/2 (2015)

Adds support for **multiplexed streams on one connection**

TCP provides a single, ordered byte stream

=> doesn’t know about multiple connections!

---

Encumbered by TCP’s semantics:
If a packet is lost, all streams suffer! 😞 😞 😞

https://www.twilio.com/blog/2017/10/http2-issues.html
HTTP/2 (2015)

Adds support for **multiplexed streams on one connection**

TCP provides a **single, ordered byte stream** => doesn’t know about multiple connections!

Encumbered by TCP’s semantics:
If a packet is lost, all streams suffer! 😞😞😞

=> Head of line blocking

https://www.twilio.com/blog/2017/10/http2-issues.html
HTTP/3 (2022): HTTP + QUIC

QUIC (RFC9000): Newer transport-layer protocol, same goals as TCP
- Supports multiple streams at once
- Various tricks to reduce message size and latency
- Integrates security by default (TLS)

- By moving multiplexing into the transport layer, can do so in a way that benefits HTTP (no head of line blocking!)
HTTP Semantics

HTTP 1.1
TLS/SSL (optional)
TCP
IPv4 / IPv6

HTTP/2
TLS 1.2+
TCP

HTTP/3
TLS 1.3
QUIC
UDP
Comparison: QUIC’s handshake
What else can we do?
Caching

Place caches throughout network
- Use locality: closer to clients => lower latency
- Improve throughput by avoiding bottleneck links

Diagram:
- Clients
- Proxy cache
- Internet
- Server
- Bottleneck

FETCH AS MUCH AS POSSIBLE FROM CACHES
How to Control Caching?

• Server sets options
  – Expires header
  – No-Cache header

• Client can do a conditional request:
  – Header option: if-modified-since
  – Server can reply with 304 NOT MODIFIED
Where to cache content?
Where to cache content?

- Client (browser): avoid extra network transfers
- Server: reduce load on the server
- Service Provider: reduce external traffic

(Eg. multiple pages on same site)
How well does caching work?

• Very well, up to a point
  – Large overlap in requested objects
  – Objects with one access place upper bound on hit ratio
  – Dynamic objects not cacheable*

• Example: Wikipedia
  – About 400 servers, 100 are HTTP Caches (Squid)
  – 85% Hit ratio for text, 98% for media

* But can cache portions and run special code on edges to reconstruct
Reverse Proxies

=> Cache close to server

- Within server's network close to it
- Distribute load, cache common resources
- => Accelerator
Reverse Proxies

• Also called Accelerators
• Can distribute load within datacenter

=> Cache close to server
Forward Proxies

- REDUCE TRAFFIC
- WORKS BEST FOR STATIC CONTENT
Forward Proxies

Typically done by ISPs or Enterprises

- Reduce network traffic and decrease latency
- May be transparent or configured
Q: CACHING & HTTPS?

CACHING SERVER needs to know THE DATA you want.

⇒ CAUSES PROBLEMS FOR HTTPS, WHICH ENCRYPTS TRAFFIC BETWEEN ENDPOINTS!

⇒ SOLUTION: HTTPS

CONNECTION "ENDPOINT" IS USUALLY THE CACHING SERVER ITSELF.

NEEDS TO USE OTHER MEANS TO DO SECURE CONNECTION TO BACKEND SERVER.
Content Distribution Networks (CDNs)

Companies that specialize in providing caching services (among other things) => Akamai, Cloudflare, ...

\[ \text{Customers push data into CDN's caches} \]
\[ \text{CDN redirects clients to their caches} \]

\[ \text{Client} \rightarrow \text{CDN} \rightarrow \text{Server} \]
Content Distribution Networks (CDNs)

Companies that specialize in providing caching services (among other things)
=> Akamai, Cloudflare, …

• Provide both forward and reverse caching

• Can also do some processing
  - TLS/RKIP
  - Video transcoding
Content Distribution Networks (CDNs)

Companies that specialize in providing caching services (among other things)

\( \Rightarrow \) Akamai, Cloudflare, \ldots

- Provide both forward and reverse caching
  - Pull: result from client requests
  - Push: expectation of high access rates to some objects
- Can also do some processing
  - Deploy code to handle some dynamic requests
  - Can do other things, such as transcoding
An Example CDN

- Forward proxies
- ISP-1
- ISP-2
- Backbone ISP
- CDN
- Clients
- Server
How a CDN works
- Eg. Best Buy sets up CDN services with CDN like Akamai
- DNS for www.bestbuy.com controlled by Akamai

When client C resolves bestbuy.com, CDN tries to find best possible cache within CDN for client
  => DNS response points to “best” server within Akamai

How you select the “best” server
Example:
  - Leverage location info for client (GeoIP, AS, …)
  - Might look up IP, do active measurements like ping/traceroute, etc.

  => DNS resolver, other caching elements are more intelligent than standard DNS server, etc.
How Akamai works

Akamai has cache servers deployed close to clients
  - Co-located with many ISPs
• Challenge: make same domain name resolve to a proxy close to the client
• Lots of DNS tricks. BestBuy is a customer
  - Delegate name resolution to Akamai (via a CNAME)
Other CDNs

- Akamai, Limelight, Cloudflare
- Amazon, Facebook, Google, Microsoft
- Netflix
- Where to place content?
- Which content to place? Pre-fetch or cache?
• **n1b.akamai.net** finds an edge server close to the client’s local resolver
  • Uses knowledge of network: BGP feeds, traceroutes. *Their secret sauce*...
dig www.bestbuy.com
;; ANSWER SECTION:
www.bestbuy.com.edgesuite.net. 21600 IN CNAME a1105.b.akamai.net.
a1105.b.akamai.net. 20 IN A 198.7.236.235
a1105.b.akamai.net. 20 IN A 198.7.236.240
  - Ping time: 2.53ms

From Berkeley, CA
a1105.b.akamai.net. 20 IN A 198.189.255.200
a1105.b.akamai.net. 20 IN A 198.189.255.207
  - Ping time: 3.20ms
Example

dig www.bestbuy.com@109.69.8.51

e1382.x.akamaiedge.net. 12 IN A 23.60.221.144

traceroute to 23.60.221.144 (23.60.221.144), 64 hops max, 52 byte packets
1 router (192.168.1.1) 44.072 ms 1.572 ms 1.154 ms
2 138.16.160.253 (138.16.160.253) 2.460 ms 1.736 ms 2.722 ms
3 10.1.18.5 (10.1.18.5) 1.841 ms 1.649 ms 3.348 ms
4 10.1.80.5 (10.1.80.5) 2.304 ms 15.208 ms 2.895 ms
5 lab-inet-r-230.net.brown.edu (128.148.230.6) 1.784 ms 4.744 ms 1.566 ms
6 131.109.200.1 (131.109.200.1) 3.581 ms 5.866 ms 3.238 ms
7 host-198-7-224-105.oshean.org (198.7.224.105) 4.266 ms 6.218 ms 8.332 ms
8 5-1-4.pearl1.boston1.level3.net (4.53.54.21) 4.209 ms 6.103 ms 5.831 ms
9 ae-4.r08.bstnma07.us.bb.gin.ntt.net (129.250.66.93) 3.982 ms 5.824 ms 4.514 ms
10 ae-6.r24.nycmy01.us.bb.gin.ntt.net (129.250.4.114) 9.735 ms 12.442 ms 8.689 ms
11 ae-9.r24.londen12.uk.bb.gin.ntt.net (129.250.2.19) 81.098 ms 81.343 ms 81.120 ms
12 ae-6.r01.mdrsp03.es.bb.gin.ntt.net (129.250.4.138) 102.009 ms 81.19.109.166 (81.19.109.166) 99.426 ms 93.236 ms 101.168 ms
14 a23-60-221-144.deploy.static.akamaitechnologies.com (23.60.221.144) 94.884 ms 92.779 ms 93.281 ms