Content Delivery: HTTP, Web Caching, and Content Distribution Networks

3035/GZ01  *Networked Systems*
Kyle Jamieson
Lecture 17

Department of Computer Science
University College London
Outline

• **Background and HTTP**
  – Cookies: client-side state
  – Limitations of HTTP/1.0
  – HTTP range requests

• **Interaction between applications and TCP**

• **Web caching**

• **Content distribution networks**
HTTP: the Hypertext Transfer Protocol

• The web’s application layer protocol

• Client/server model over TCP
  – *Client*: browser that requests, receives, web objects
  – *Server*: web server responds to requests with data
    • Accepts connections on TCP port 80

• HTTP itself is “stateless:” the protocol maintains no information about past client requests
  – Design goal: don’t burden web servers with state maintenance
  – Cookies added later to carry state client-side (scalable)

• Metadata plays a key role in design and operation
An HTTP/1.0 exchange

**Request:**
GET /index.html HTTP/1.0

**Response:**
HTTP/1.0 200 OK
MIME-Version: 1.0
Server: CERN/3.0
Date: Thu, 03 Dec 2009 16:15:53 GMT
Content-Type: text/html
Content-Length: 22631
Last-Modified: Wed, 11 Nov 2009 09:10:45 GMT

<p class="contact_new">Computer Science Department
University College London – Gower Street – London –
WC1E 6BT – <img src="/images/phone.gif" width="13" height="9" id="phone" alt="Telephone:"></p> +44 (0)20 7679 7214 – Copyright © 1999-2007 UCL</p>
Cookies: Client-side state

- Recall: the HTTP protocol is stateless

- **However:** state is useful for session authentication, online sessions (online shopping, banking, etc.)

- **Idea:** **client** keeps small pieces of state *(cookies)*
  - Received from server in responses, sent to server in requests
  - At least two possibilities:
    - All state may be encoded into cookie stored at client
    - Cookie indexes state in a database on the server (online shopping)

- **Privacy issue:** often user is unaware of cookies
  e.g.: search engine returns link `http://ad.doubleclick.net/x26362d2/www.foo.com/foo.html`
Cookies allow content personalization

Client

HTTP GET

HTTP/1.0 200 OK; Set-cookie: 1678

HTTP GET; Cookie: 1678

HTTP/1.0 200 OK

HTTP GET; Cookie: 1678

HTTP/1.0 200 OK

One week later

Server

ucl.ac.uk

Create id 1678

Backend database

Lookup id 1678

cookies.txt

ucl.ac.uk: 1678

Lookup id 1678
Problems with HTTP/1.0

1. Poor use of TCP, especially for short responses
   - Recall: three-way handshake to open TCP connection, four packets to close TCP connection
     - HTTP message often fits in $\approx 10$ packets $\rightarrow$ overhead
     - Don’t get past slow-start, so don’t use available bandwidth
   - For each embedded image: open a new TCP connection and send another HTTP GET

2. Lack of support for web caching

3. Lack of bandwidth optimization
   - Range requests
   - Data compression
HTTP/1.1 bandwidth optimization

- Data compression
  - Image files pre-compressed, but much other content is not
  - [1997]: compression would save 40% of bytes sent via HTTP
  - Client uses Content-Encoding header in HTTP GET to signal which encodings it can decompress, and which it prefers

- Range requests
  - Problem: client wants to resume an incomplete download, view certain pages of a long document
  - Need to retrieve just a small section of a resource

GET bigfile.html HTTP/1.1
Range: 2000-3999

HTTP/1.1 206 Partial Content
Date: Thu, 10 Feb 2006 20:02:06 GMT
Last-Modified: Wed, 9 Feb 2006 14:58:08 GMT
Content-Range: bytes 2000-3999/100000
Content-Length: 2000
Content-Type: text/html
Outline

• Background and HTTP

• **Interaction between applications and TCP**
  – Connection establishment and slow start
  – Web transfers, HTTP/1.1 persistent connections
  – Interactive applications: Nagle’s algorithm

• Web caching

• Content distribution networks
TCP connection establishment delay

- Recall TCP retransmit timeout: 
  \[ \text{RTO} = \text{RTT} + 4 \times \text{variance} \]
  - Initial RTO = 3 seconds
  - Double RTO on timeout
- Avoids unnecessary SYN retransmissions if RTT is large
- Frustrated users hit stop button (terminate TCP connections) and reload button
- **Ongoing issue:** Lowering initial RTO is a topic of active discussion in the IETF (May ‘11)
Effect of lowering initial RTO

- Today on the Internet
  - ≈ 2% TCP flows lose first SYN
    - Current proposal: Retransmit SYN after 1 sec
    - Lower penalty for SYN loss from 3 secs to one sec
  - ≈ 2% TCP flows’ RTT > 1 sec
    - Why? 802.11 MAC layer, slow dialup line, slow VPN connection
    - Spurious SYN retransmission
- Tradeoff between spurious SYN retransmission and user wait
Delay in the middle of a web transfer

• Now, the TCP connection is open and in slow start
• Recall TCP slow start: increase sender cwnd by a packet size for each received acknowledgement
  – Short HTTP transfers spend most of their time in slow start
• Small cwnd means that after a loss, receiver unlikely to generate three dup ACKs
  – Retransmission timeouts are more likely

![Sender sequence number (bytes)](image)
An HTTP/1.0 page fetch (Obsolete)

- Fetch an 8.5 Kbyte page with 10 embedded objects, most < 10 Kbyte
- One HTTP transaction/fetch, open and close a new TCP connection each time
- Stay in slow start, except for the large object
Improvement: Persistent HTTP connections

- Keep TCP connection open for many HTTP transactions
  - Reduce TCP connection setup and teardown overhead
  - Remove slow start performance problems (cwnd grows)

- *Keep-Alive* mechanism in HTTP/1.0 (a persistent TCP connection is the default in HTTP/1.1)

```
GET /index.html HTTP/1.0
Connection: Keep-Alive

reply from server
GET image1.jpg HTTP/1.0
Connection: Keep-Alive

reply from server
GET image2.jpg HTTP/1.0
Connection: Keep-Alive

GET /index.html HTTP/1.1

reply from server
GET image1.jpg HTTP/1.1

reply from server
GET image2.jpg HTTP/1.1
```
Persistent connections avoid unnecessary slow starts

- Fetch an 8.5 Kbyte page with 10 embedded objects, most < 10 Kbyte
- Leave TCP connection open after server response, next HTTP request reuses
- Only incur one slow start, but takes an RTT to issue next request
HTTP request pipelining

- Normally client only issues HTTP requests after server’s response
- Clients using HTTP pipelining issue multiple requests without waiting for responses
- Advantages
  - Reduces latency of page loading
  - Can reduce overhead, packing multiple requests into one packet

```
GET /index.html HTTP/1.1
GET image1.jpg HTTP/1.1
GET image2.jpg HTTP/1.1
```

```
GET /index.html HTTP/1.1
GET image1.jpg HTTP/1.1
GET image2.jpg HTTP/1.1
```
Pipelined requests overlap RTTs

- Fetch an 8.5 Kbyte page with 10 embedded objects, most < 10 Kbyte
- Send multiple HTTP requests simultaneously
- May or may not use same TCP connection - fairness issues
Outline

• Background and HTTP
• Interaction between applications and TCP
  – Connection establishment and slow start
  – Web transfers, HTTP/1.1 persistent connections
  – Interactive applications: Nagle’s algorithm
• Web caching
• Content distribution networks
Interactive applications: When *exactly* should TCP send?

- Server writing data in pieces of size < MSS into a TCP sender socket buffer
- Common case: cwnd ≥ maximum segment size (MSS)
- Transport layer problem: *when to transmit a segment?*
  - Early TCP: send immediately when data arrives
    - Wastes capacity, sending small packets (“tinygrams”)
  - Send when reach MSS? Break ssh, remote desktop, other interactive apps
Nagle’s algorithm

- Proposed in RFC 896, in 1984
- Delay sending until have MSS bytes of data to send, but
- Use the ACK clock to trigger small packet transmission
  - Low RTT connections: ACK comes back before next keystroke
  - High RTT connections: amortize keystrokes, avoid tinygrams

*When application generates new data:*

```
if available data ≥ MSS and cwnd ≥ MSS then
  send a full segment
else if unacked data in flight then
  buffer new data until ACK arrives
else send immediately
```

*On ACK receipt:*

send pending data (subject to cwnd)
Nagle’s algorithm on a WAN

- Various amounts of data from slip (client) to vangogh: 1, 1, 2, 1, 2, 2, 3... bytes

- Coalescing data until previous data acknowledged

- Segments 14, 15 appear to contradict Nagle
  - 14 is in response to 12
  - 15 is in response to 13

[Stevens, TCP/IP Illustrated, vol. 1]
Nagle’s algorithm at a web server

- Suppose MSS=1460 bytes, and HTTP header length = 25 bytes, HTTP response length = 1000 bytes, cwnd large

  
  - Client
  - Server

  Server with Nagle on, separate write calls

  write(1:25)
  write(26:1025)

  Wasted time

  Server with Nagle on, one write call

  write(1:1460)
  1461:2920
  2921:2925

  Wasted time

  (5 bytes remain)

- HTTP server should disable Nagle’s algorithm (setsockopt(...,TCP_NODELAY,...)) and send data in one write
Outline

• Background and HTTP
• Interaction between applications and TCP
• Web caching
  – Conditional GET
  – Cache coherency
  – Web cache hierarchies
• Content distribution networks
Web caching: Introduction

- Why cache web pages?
  1. Reduce load on web servers ("origin" servers)
     - Persistent load
     - Flash crowds ("Slashdot effect")
  2. Reduce Internet congestion
  3. Reduce bandwidth consumption in the Internet
     - Incentive: reduce ISP peering costs
  4. Improve client fetch latency
  5. Improve availability of web pages for clients

[Jung, Krishnamurthy, Rabinovich in WWW '02]
Web caching: Mechanism

- Web proxy is somewhere on path from client to origin server
- User configures browser to access web via proxy
- Browser sends all HTTP requests to the cache
  - Object in cache: cache returns object in HTTP response
  - Otherwise: cache requests object from origin server, then returns object to client
- Proxies may be chained: requests go from client to server via more than one proxy
More about web caching

- Cache acts as both client and server
- Typically cache is installed by ISP (university, company, residential ISP)
- Internet dense with caches: enables “poor” content providers to effectively deliver content (but so does peer-to-peer file sharing)
Caching example

Assumptions
• Avg. object size: 100 Kbits
• Avg. request rate from institution’s browsers to origin servers: 15/sec
• Institutional router – origin server RTT: 2 sec.

Consequences
• LAN Utilization: 15%
• Access link utilization: 100%
• Total delay = Internet delay + access delay + LAN delay
  = 2 sec. + minutes + milliseconds
Caching example: Provisioning

• Increase bandwidth of access link to, say, 10 Mbps

Consequence
• LAN utilization: 15%
• Access link utilization: 15%
• Total delay = Internet delay + access delay + LAN delay
  = 2 sec. + millis. + millis.
• Often a costly upgrade
Caching example: Web cache

- Install a web cache
  - Suppose hit rate is 0.4

**Consequence**
- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- Utilization of access link reduced to 60%, resulting in negligible delays (say 10 milliseconds)
- Total average delay = Internet delay + access delay + LAN delay = \(0.6 \times (2.01)\) secs + \(0.4\) milliseconds < 1.4 secs
Where to place caches?
☞ Where are Internet bottlenecks?
1. Network link from Hosting AS to server
2. Access line from client to ISP
3. Client’s ISP to backbone AS
4. Client’s Internal AS
5. Peering points
6. The web server itself

- Hosting service AS edge (2.) ✔, (6.) ✔
- Client’s premises (1.) ✔
- An ISP’s AS (3.) ✔
Cache consistency

• Definition: Cached objects match corresponding objects on origin server

• HTTP/1.1 has mechanisms for ensuring freshness:
  – Expires header on server responses
  – Cache control header on server responses
    Cache-control: no-cache
      • Forces caches to resubmit request to origin server every time
    Cache-control: no-store
      • Forbids caches from storing content under any conditions
    Cache-control: max-age=[seconds]
      • Similar to expires header, relative time
Conditional GET

• **Goal:** Don’t send object if cache has up-to-date cached version

• Cache: specify date of cached copy in HTTP request
  – If-modified-since: <date>

• Server: response contains no object if cached copy is up-to-date:
  – HTTP/1.0 304 Not Modified

• HTTP request
  If-modified-since: <date>

• HTTP response
  HTTP/1.0 304 Not Modified

• HTTP request
  If-modified-since: <date>

• HTTP response
  HTTP/1.0 200 OK
  <data>
Cooperative caching

• Can arrange caches together in a cluster
• How to locate an item in a cluster of caches?
1. Always local (caches in cluster don’t cooperate); results in duplication
2. Primary plus multicast
   – Uses inter-cache protocol such as Internet Cache Protocol (ICP)
3. Hashing
   – Direct access to resource
   – Changing the hash function can result in objects moving about
4. Consistent hashing
Outline

• Background and HTTP
• Interaction between applications and TCP
• Web caching

• Content distribution networks
  – Mechanism: client DNS redirection
  – Consistent hashing for server selection
  – Operation under failure
Content distribution networks

• Challenge ca. 2002: *How to reliably deliver large amounts of content to users worldwide?*
  – Flash crowds overwhelm web server, access link, or backend database infrastructure
  – More rich content: audio, video

• Possible solutions
  – Web caching: *dynamic content and diversity causes low proxy hit rates (25–40%)*
  – Multihoming for the server: *BGP takes minutes to converge upon route failure*
Content distribution networks

- Replicate content at the edge
  - Deploy thousands of content distribution network (CDN) edge servers at many ISPs
  - Push content to edge servers ahead of time
  - Avoid impairments (loss, delay) of using long paths
  - Solve “flash crowds” problem

- Which CDN server?
  1. Nearest server to client
  2. Likely to have content
  3. Available (load, bandwidth)
  4. Adapt choice quickly (secs.)
Sending clients to the CDN for content

- Web client fetches web page via HTTP GET, as usual
- Content provider rewrites content links to point to CDN
  - Browser will issue HTTP GETs for rich content to the CDN edge servers instead of the content server
  - CDN companies provide automated tools to rewrite links
An “Akamized” URL (ARL)

Serial number: Defines content served from same set of servers
Type code: $f$ = static TTL, $6$ = If-Modified-Since, $7$ = MD5 hash
  – Various ways of ensuring freshness of object served
Content provider code (CPC): Unique ID for content provider
Object data: $f$ = time, $6$ = “000”, $7$ = MD5 hash value
Absolute URL: Used to retrieve content from provider

```
http://a764.g.akamai.net/f/764/16742/1h/
```

```
www.1800flowers.com/800f_assets/jet/website/
```

```
images/flowers/flowers/banners/hp/hpb/
```

```
btn_findagift.gif"
```

• Note: Serial number is the only input to DNS name resolution
Mapping requests to CDN servers

• Goal: find the CDN server **nearest to the client**
  – Establish BGP peering sessions with Internet border routers → coarse-grained AS map of Internet
  – Combine with live traceroute, loss measurement data between CDN servers
  – **Result: a map of the Internet**

• Finding an **available** CDN server
  – Server health, service requested, server load balancing (CPU, disk, network), and network condition
  – Agents simulate user behavior, downloading objects, and reporting failure rates and service times
DNS-based redirection

- Two levels of DNS indirection
  1. Akamai top-level nameservers (TLNSs)
     - Locations: US (4), Europe (4), Asia (1)
     - TLNSs return eight LLNSs in three different regions
       - Chosen to be close to the requesting client (use the map)
       - Handles complete failure of any two particular regions
  2. Akamai low-level nameservers (LLNSs)
     - Point to Akamai edge servers, which serve content
     - Do most of the load-balancing
DNS resolution

1. End user
2. Browser cache
3. OS
4. xyz.com
5. 10.10.123.5
6. ak.xxyz.com
7. a212.g.akamai.net
8. akamai.net
9. 15.15.125.6
10. g.akamai.net
11. 20.20.123.55
12. a212.g.akamai.net
13. 30.30.123.5
14. Local nameserver
15. OS
16. End user

xyz.com’s nameserver

Allocate and select cluster

Select servers within cluster

Akamai TLNS

Akamai LLNS

[Slide: Bruce Maggs]
Akamai DNS redirection

Web clients’ network

Any network

Edge server cluster

Select server

LLNS

TTL 20s

Content

End user

Select cluster

TTL 30−60m

TLNS

Content server

(news.bbc.co.uk)

Content provider’s network

Network close to client

HTML

Content server
Akamai Cluster

Servers pool resources

- RAM
- Disk
- Throughput
Hashing

• Universe $U$ of all possible objects, set $S$ of servers
  – **Object**: Web content objects
  – **Server**: Edge content server

• Hash function $h: U \rightarrow S$ assigns objects to servers
  – E.g., $h(x) = [ax + b \ (\text{mod } p)] \ (\text{mod } |S|)$, where
    • $p$ is a prime integer, and $p > |S|$
    • $a, b$ are constant integers chosen uniformly at random from $[0, p - 1]$
    • $x$ is an object’s serial number
Difficulty: Changing number of servers

\[ h(x) = x + 1 \pmod{4} \quad \text{Add one machine:} \quad h(x) = x + 1 \pmod{5} \]
Consistent hashing

• Low-level DNS servers act independently and may have different ideas about how many and which servers are alive

• Who uses it?
  – Akamai
  – amazon.com: Dynamo data store
  – Last.fm music website
  – Akamai CDN
  – Chord P2P network
Consistent hashing

Map both objects and servers onto the unit circle.

For objects with serial number \( x \):
\[
h(x) = \text{successor}( \left[ ax + b \ (\text{mod} \ p) \right] / p )
\]

For servers with id number \( x' \):
\[
h(x') = \left[ ax' + b \ (\text{mod} \ p) \right] / p
\]
Properties of consistent hashing

- **Balance:** Objects are assigned to buckets “randomly”
- **Locality:** When a server is added/removed, the only objects affected are those that are/were mapped to the bucket
- **Load:** Objects are assigned to buckets evenly, even over a set of views
  - Can be improved by mapping each server to multiple places on the unit circle
- **Spread:** An object should be mapped to a small number of servers over a set of views
  - Can choose different $a, b$ to make a new hash function
  - Use multiple hash functions to map object to multiple servers
Adding and removing servers

- Each server is responsible for a different section of the ring
- Adding a server
  - Only need to move content between predecessor server and new server (segment AC)
- Removing a server
  - Clean shutdown: Server pushes content to its successor
  - Failures: lookup a different server, and push content to successor in background
Hardware/server failures

- Linux and Windows servers with large RAM and disk capacity
- Example failures
  1. Memory modules jumping out of their sockets
  2. Network cards screwed down but not in slot
  3. Hardware component failures: HDD, motherboard, etc.
  4. Others
Operation under failure

- **Network X fails**
  - No impact on already-selected LLNSs
  - Different TLNS next time
  - Mapping fails over

- **Server in cluster A fails**
  - Detection using heartbeats
  - Another server ("buddy") proxy-ARPs (immediately)
  - LLNS removes failed server (30−60 min)

- **Transit network B fails**
  - Route using intermediate server in transit network E

- **Network A or LLNS A fails**
  - Client’s local NS will use another LLNS (30−60 min)
Low-level DNS load balancing

Random permutations of servers

Why? To spread load for one serial number.
LLNS DNS-based load balancing

• Input: which servers have which content (from consistent hashing), server load, proximity to client (from DNS request)

• Consider edge server A. Two thresholds:
  1. Load > low threshold: add servers to *distribute load* (more hash functions to consistent hash ring)
  2. Load > high threshold: *shed load* on that server (remove that hash function from consistent hash ring)