416 Distributed Systems

January 27, 2022

Making the web fast:

SPDY/HTTP2.0, CDNs

Consistent hashing

Special thanks to Sophia Wang for some slides

Outline

- Last time: distributed file systems
- Today: the web

- Problem with HTTP 1.1
- SPDY and HTTP2.0
- DNS Design (covered in 317)
- Content Distribution Networks
- Consistent hashing

Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes are heavy-tailed
- Embedded references
- This plays havoc with performance. Why?

Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes are heavy-tailed
- Embedded references
- This plays havoc with performance. Why?

- Lots of small objects & TCP
  - 3-way handshake
  - Lots of slow starts
  - Extra connection state

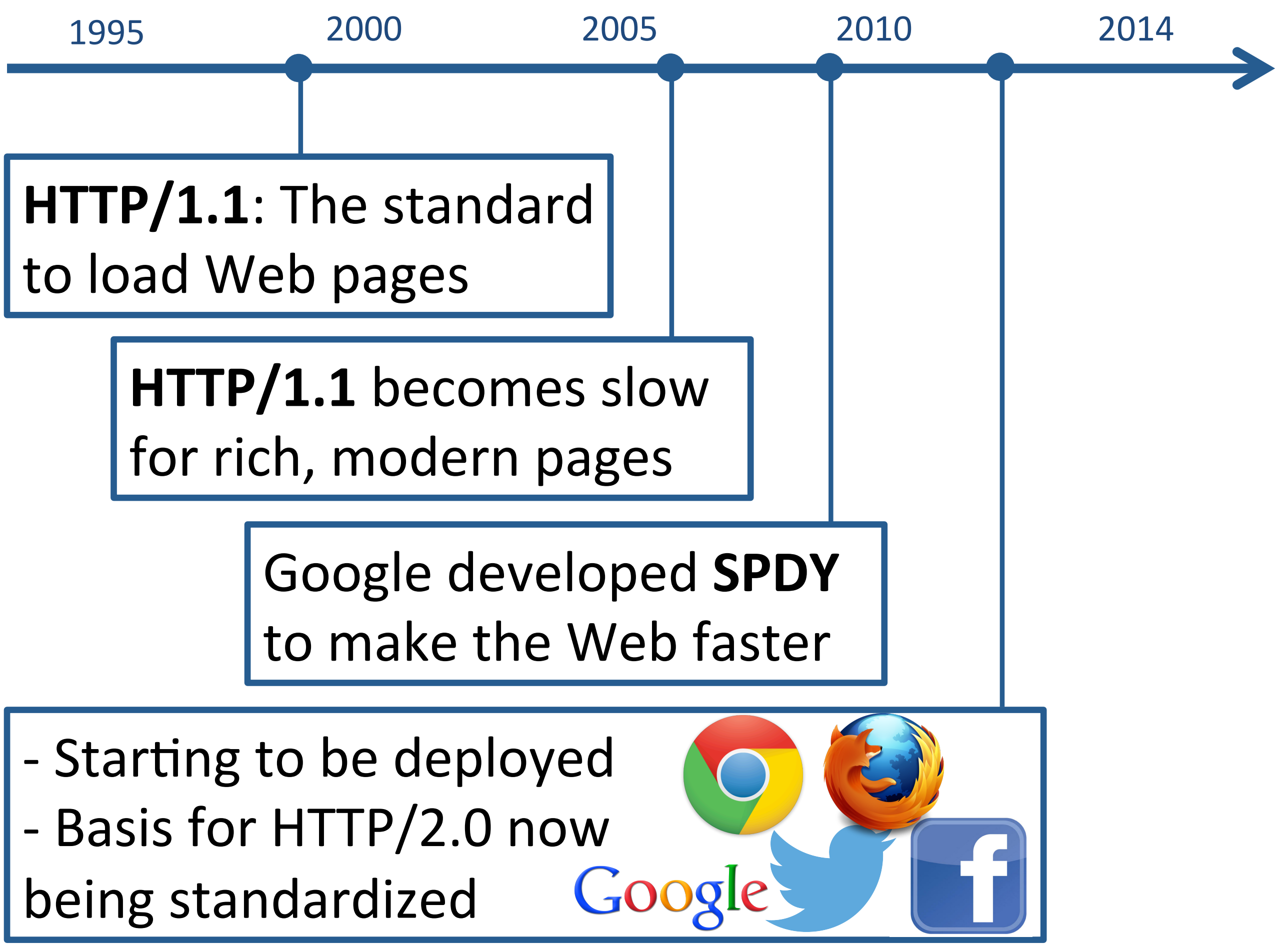
Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes are heavy-tailed
- Embedded references
- This plays havoc with performance. Why?

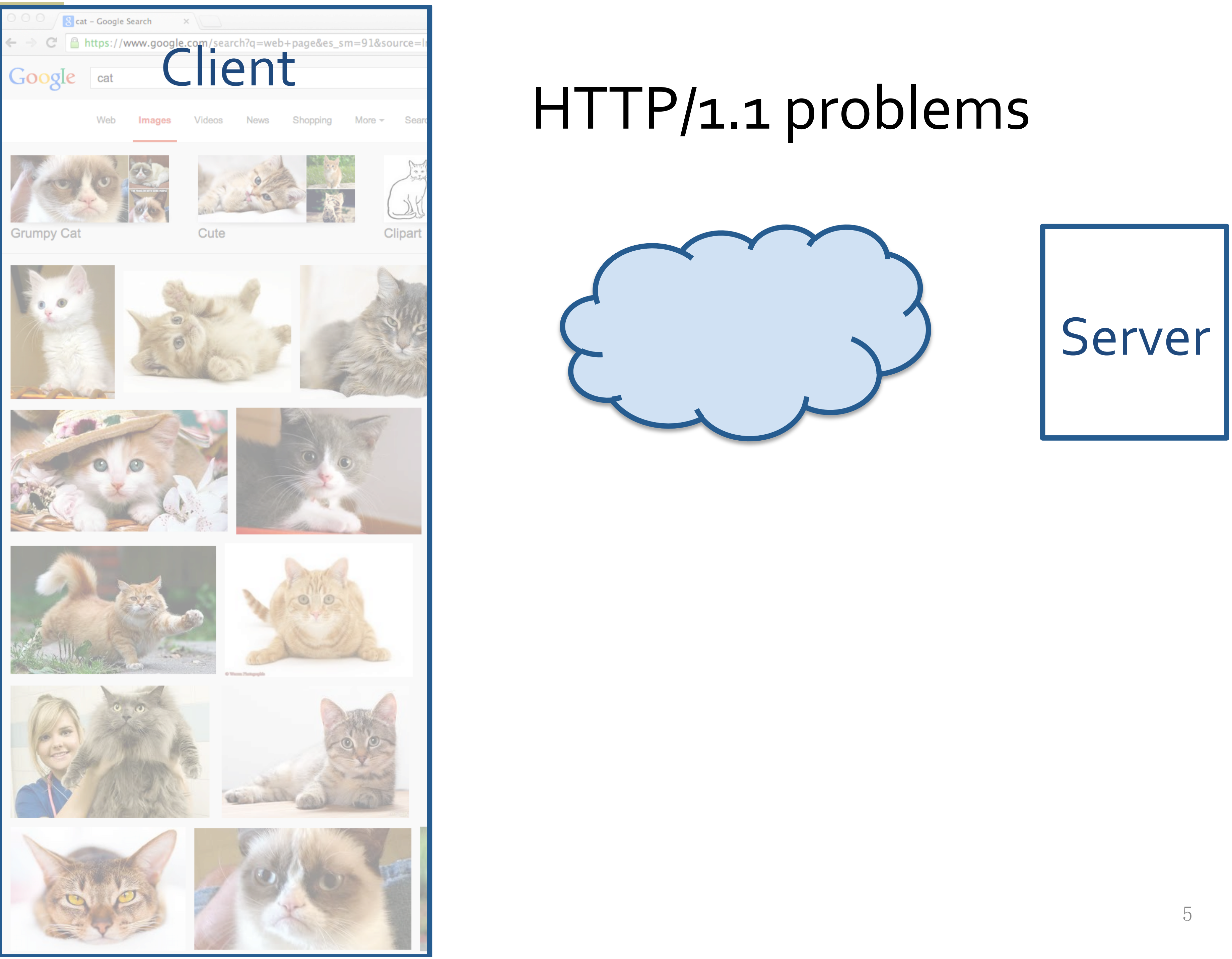
- Lots of small objects & TCP
  - 3-way handshake
  - Lots of slow starts
  - Extra connection state

- Solutions?
  - New protocol!
    - (SPDY -> HTTP 2.0)
  - Web caches
  - CDNs

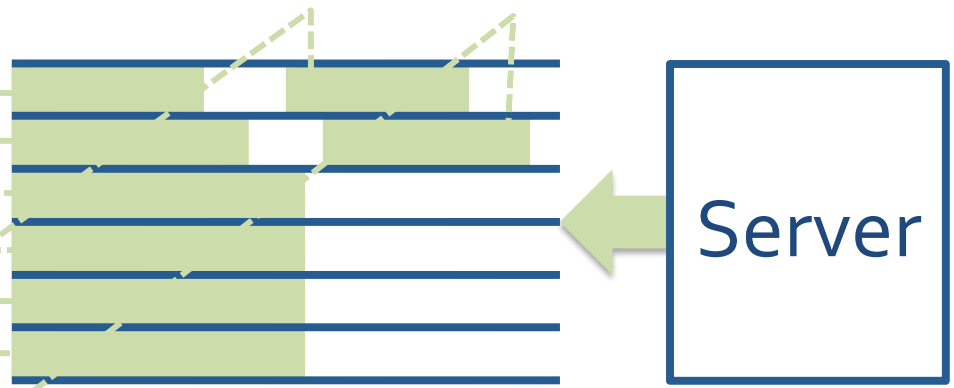
HTTP evolution



HTTP/1.1 problems

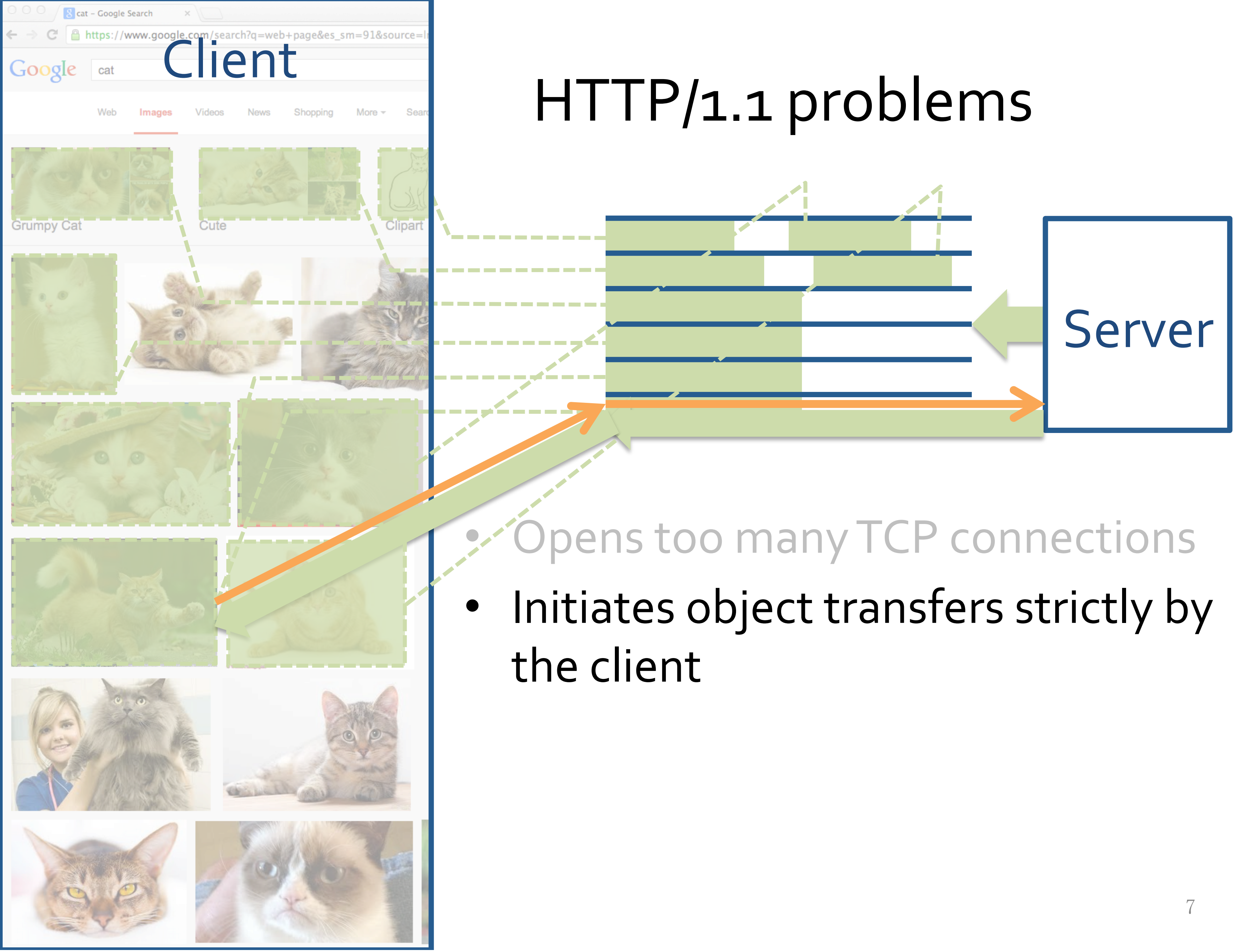


HTTP/1.1 problems



- Opens too many TCP connections

HTTP/1.1 problems



- Opens too many TCP connections
- Initiates object transfers strictly by the client

HTTP/1.1 problems

- Opens too many TCP connections
- Initiates object transfers strictly by the client
- Compresses only HTTP payloads, not headers

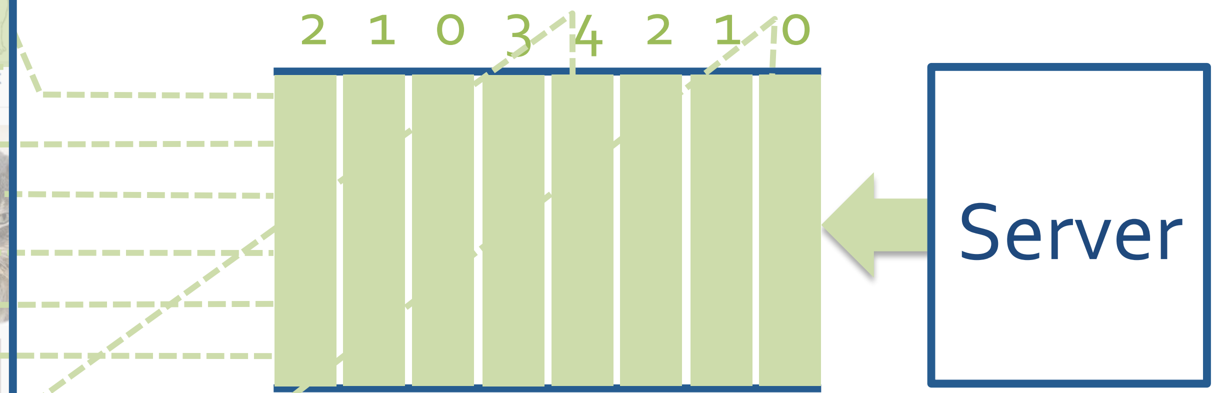
HTTP/1.1 problems

HTTP/1.1 200 OK\r\n

SPDY is proposed to address these issues

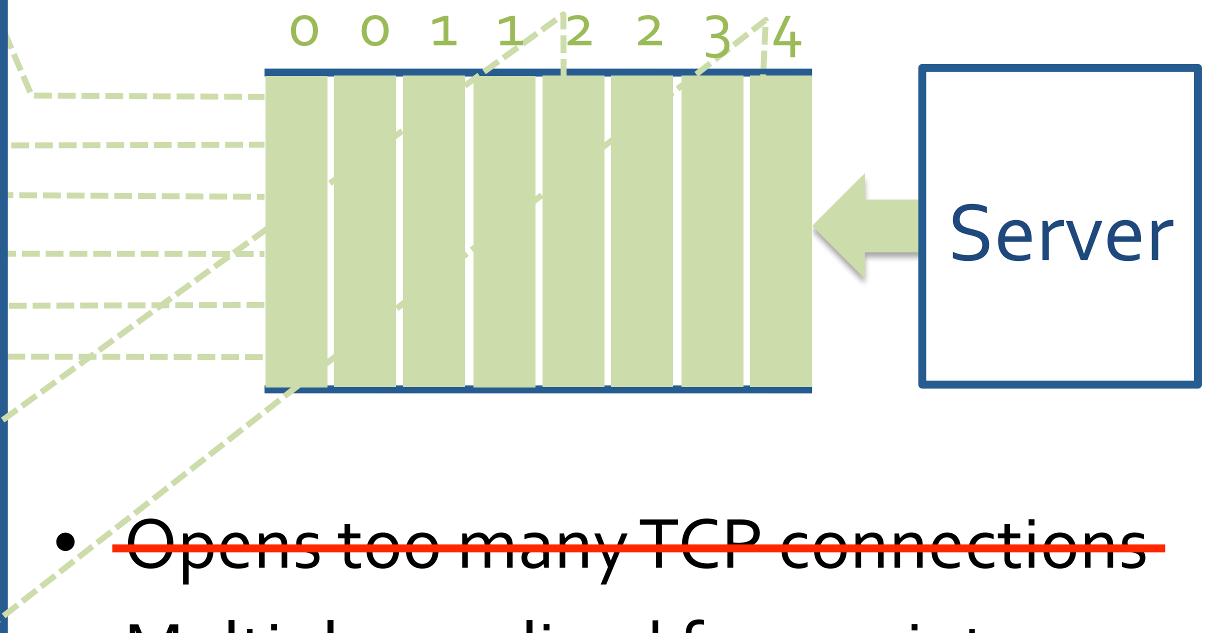
- Opens too many TCP connections
- Initiates object transfers strictly by the client
- Compresses only HTTP payloads, not headers

SPDY



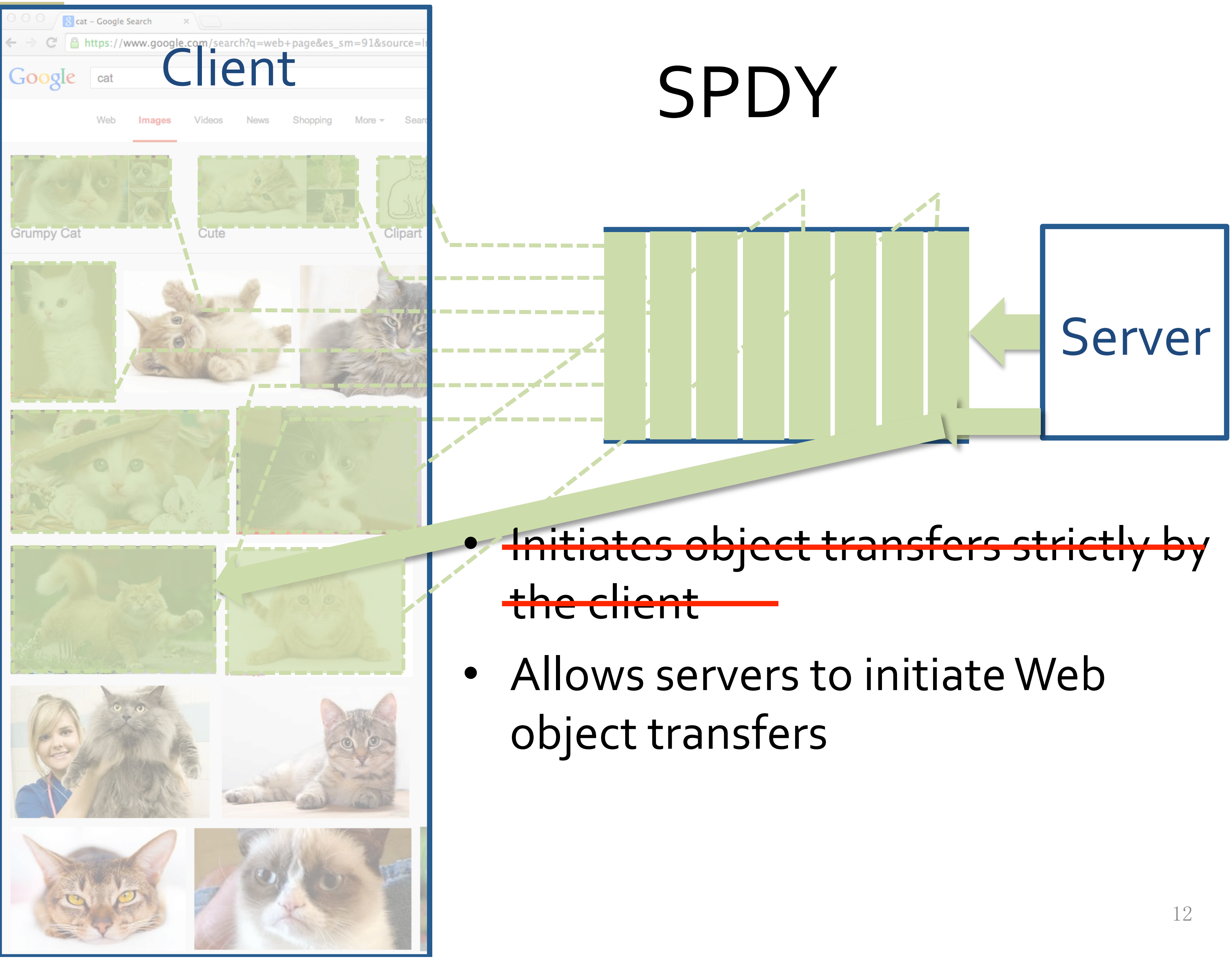
- ~~Opens too many TCP connections~~
- Multiplexes sliced frames into a single TCP connection

SPDY



- ~~Opens too many TCP connections~~
- Multiplexes sliced frames into a single TCP connection
- Prioritizes Web objects

SPDY



- ~~Initiates object transfers strictly by the client~~
- Allows servers to initiate Web object transfers

SPDY

- ~~Compresses only HTTP payloads, not headers~~
- Compresses both HTTP payloads and headers

HTTP evolution: SPDY->HTTP 2.0 !

1995 | 2000 | 2005 | 2010 | 2014 | 2015 | 2021

HTTP/1.1: The standard to load Web pages

HTTP/1.1 becomes slow for rich, modern pages

Google developed **SPDY** to make the Web faster

- Starting to be deployed
- Basis for HTTP/2.0 now being standardized

Google

2015: HTTP/2 Introduced

G. drops SPDY support

2021: 50% web HTTP/2

Outline

- Problem with HTTP 1.1
- SPDY and HTTP2.0
- DNS Design (covered in 317)
- Content Distribution Networks
- Consistent hashing

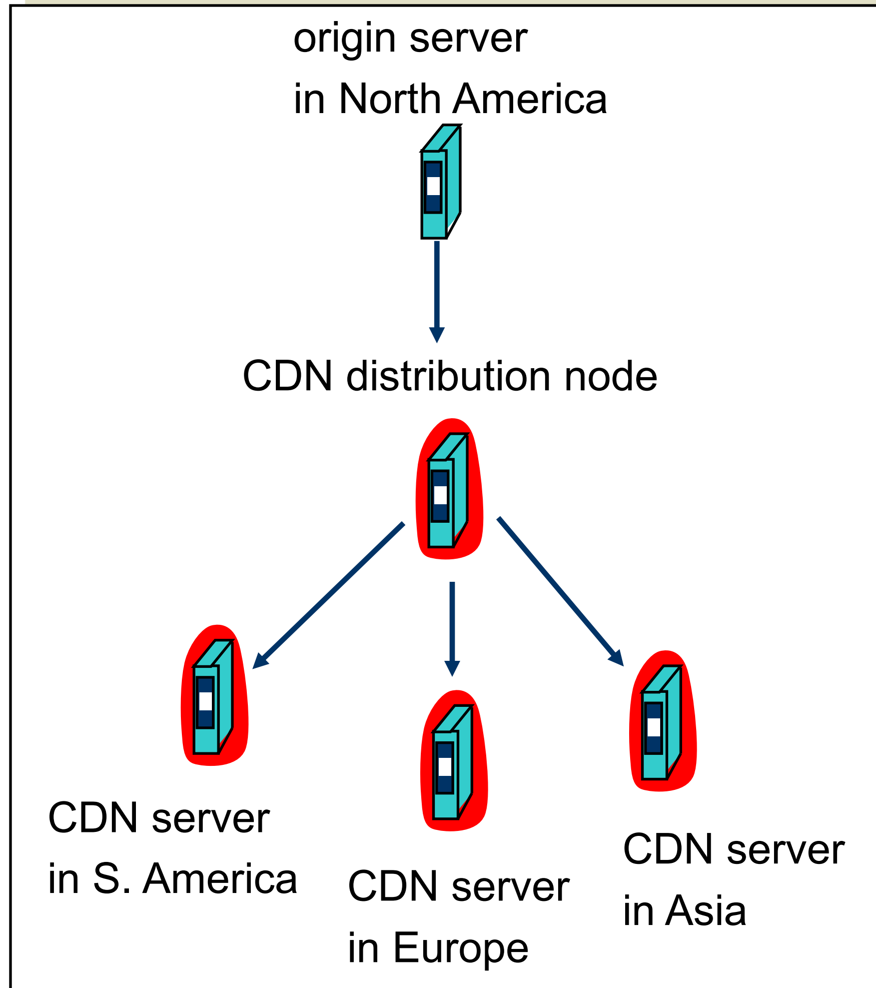
Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes are heavy-tailed
- Embedded references
- This plays havoc with performance. Why?
- Solutions?
  - New transport (SPDY)
  - Web caches
  - CDNs: redesign delivery

- Lots of small objects & TCP
  - 3-way handshake
  - Lots of slow starts
  - Extra connection state

Content Distribution Networks (CDNs)

- The content providers are the CDN customers.
- Content replication
- CDN company installs hundreds of CDN servers throughout Internet
  - Close to users
- CDN replicates its customers' content in CDN servers. When provider updates content, CDN updates servers
- An example of how a distributed system can improve *latency*



Content Distribution Networks & Server Selection

- Replicate content on many servers

- CDN distributed design challenges
  - How to replicate content
  - Where to replicate content
  - How to find replicated content
  - How to choose among known replicas
  - How to direct clients towards replica

Server Selection

- Which server?
  - Lowest load → to balance load on servers
  - Best performance → to improve client performance
    - Based on Geography? RTT? Throughput? Load?
  - Any alive node → to provide fault tolerance
- How to direct clients to a particular server?
  - As part of routing → anycast, cluster load balancing
  - As part of application → HTTP redirect
  - As part of naming → DNS

Application Based

- HTTP supports a simple way to indicate that Web page has moved (30X responses)
- Server receives GET request from client
  - Decides which server is best suited for particular **client** and **object**
  - Returns HTTP redirect (to the client) to that server
- Can make informed application specific decision
- May introduce additional overhead →
  multiple connection setup, name lookups, etc.

Naming Based

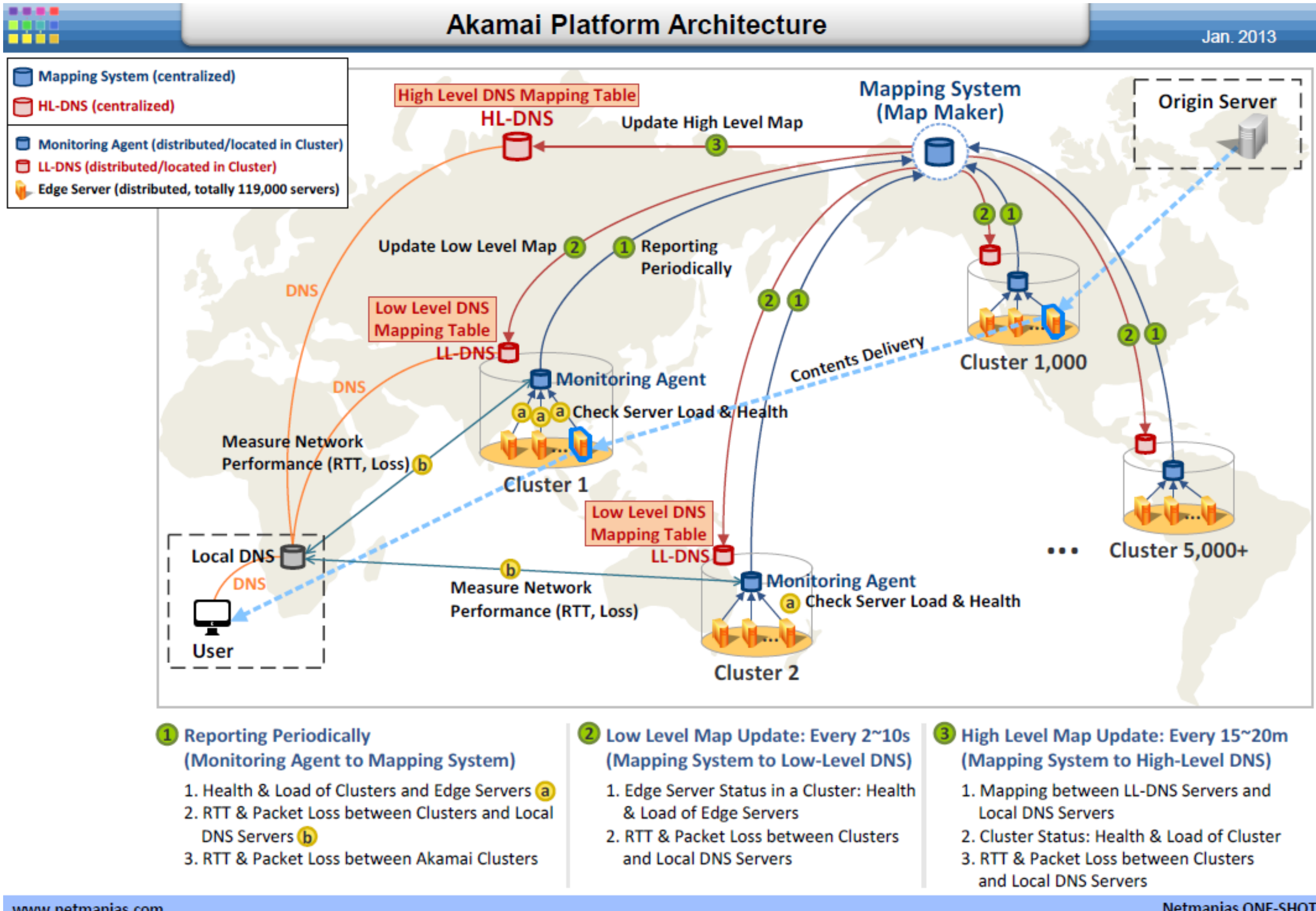
- Client does name lookup for service
- Name server chooses appropriate server address
  - DNS A-record returned is “best” one for the client
- What information can name server base decision on?
  - Web server load/location → must be collected
  - Information in the name lookup request
    - Name service client → typically the local name server for client (not the client itself, which means not aware of the app making the DNS request)

How Akamai Works

- Akamai only replicates static content (*)
- Modified name contains original file name
- Akamai server is asked for content
  - First checks local cache
  - If not in cache, requests file from primary server and caches file

* (At least, the version we're talking about today. Akamai actually lets sites write code that can run on Akamai's servers, but that's a pretty different beast)

Akamai overview



How Akamai Works

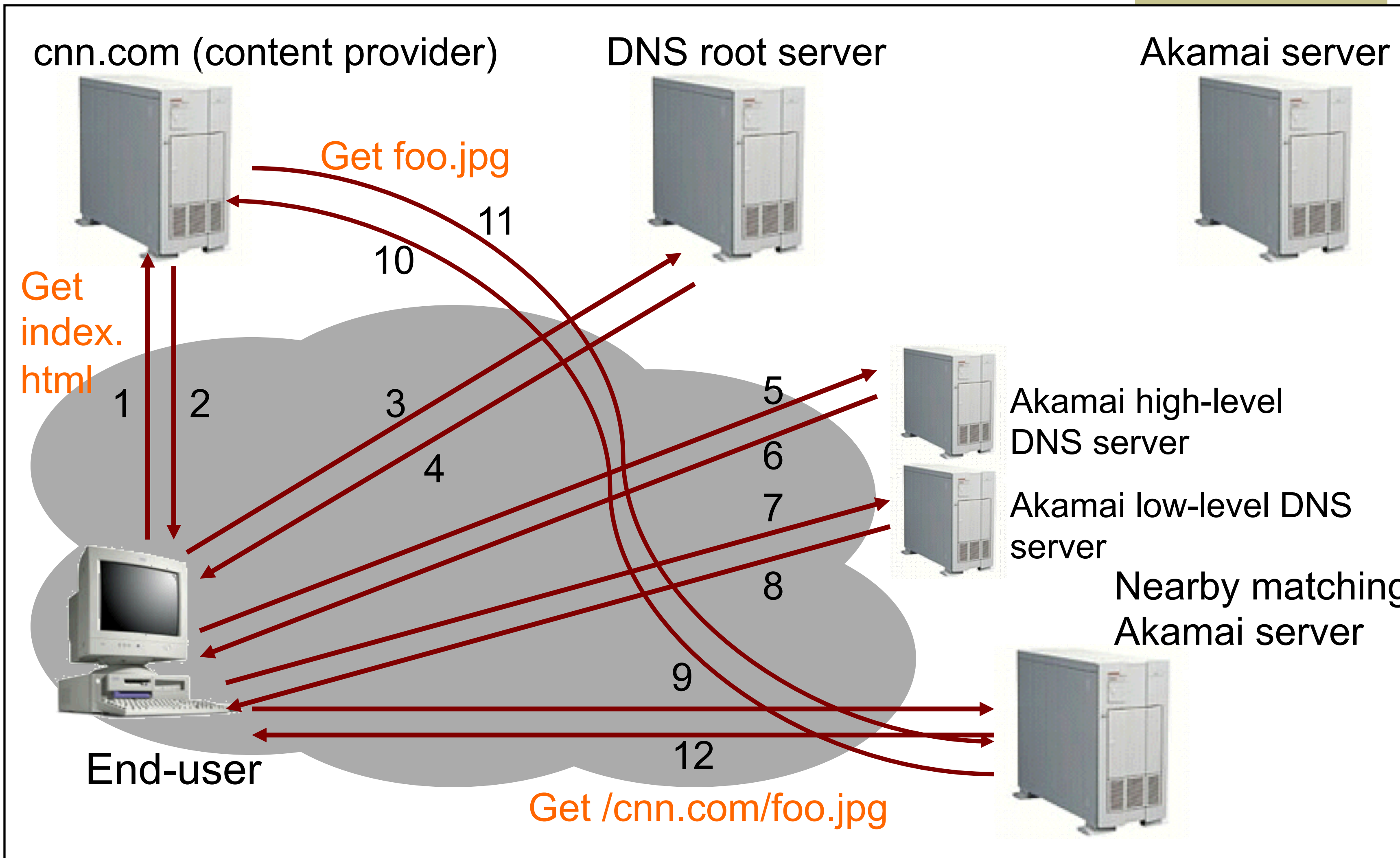
- Clients fetch html document from primary server
  - E.g. GET index.html from cnn.com
- URLs for replicated content are replaced *in html*
  - E.g. <img src=“http://cnn.com/af/x.gif”>
  - replaced with

    <img src=“http://**a73.g.akamaitech.net**/7/23/cnn.com/af/x.gif”>
- Client is forced to DNS resolve aXYZ.g.akamaitech.net hostname

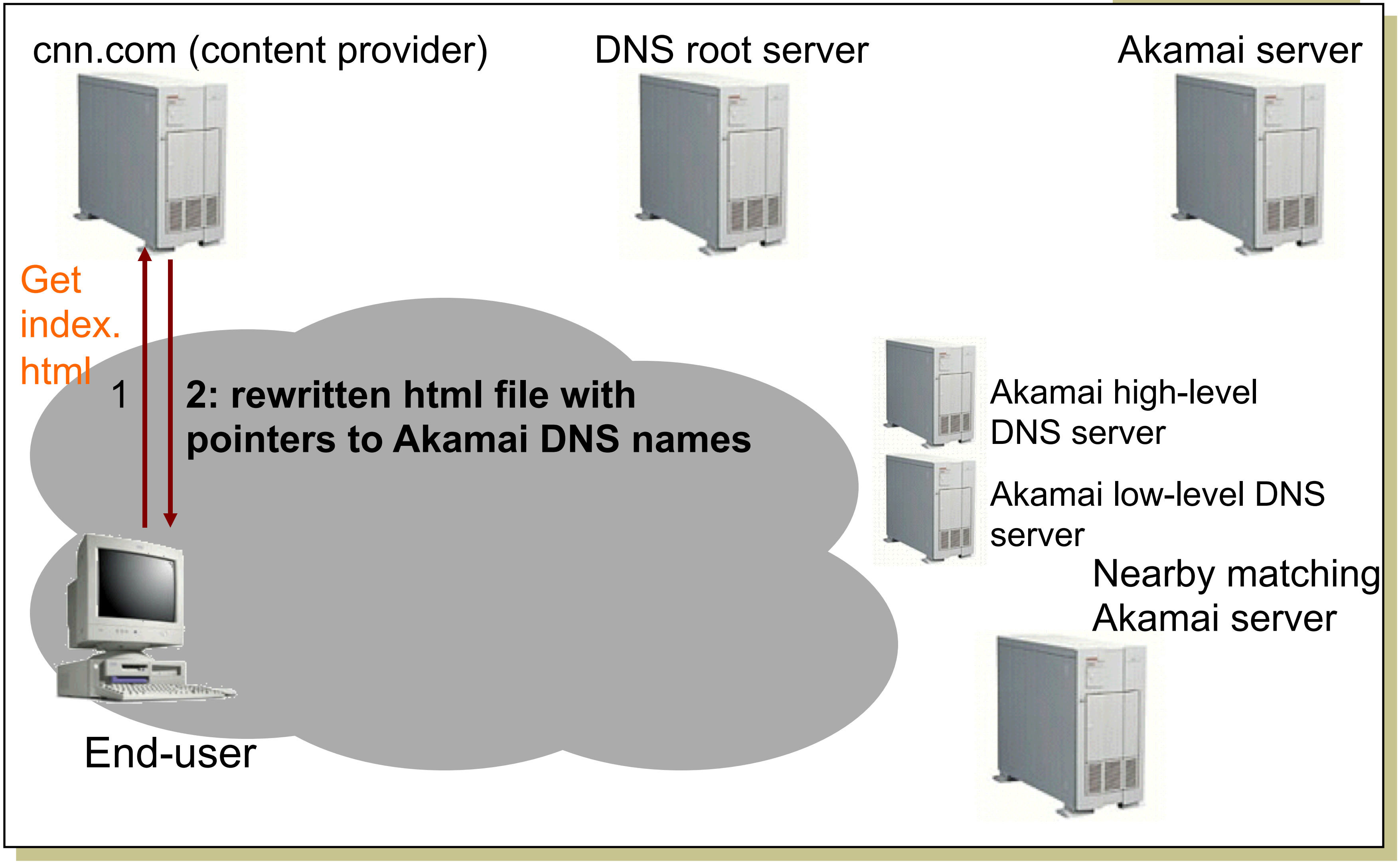
How Akamai Works

- Root server gives NS record for akamai.net
- Akamai.net name server returns NS record for g.akamaitech.net
  - Returned name server chosen to be in **region** of client's name server
  - DNS TTL is large
- G.akamaitech.net nameserver chooses server in region
  - Should try to chose server that has file in cache - How to choose?
  - Uses object (aXYZ) name and hash
  - DNS TTL is small → why?

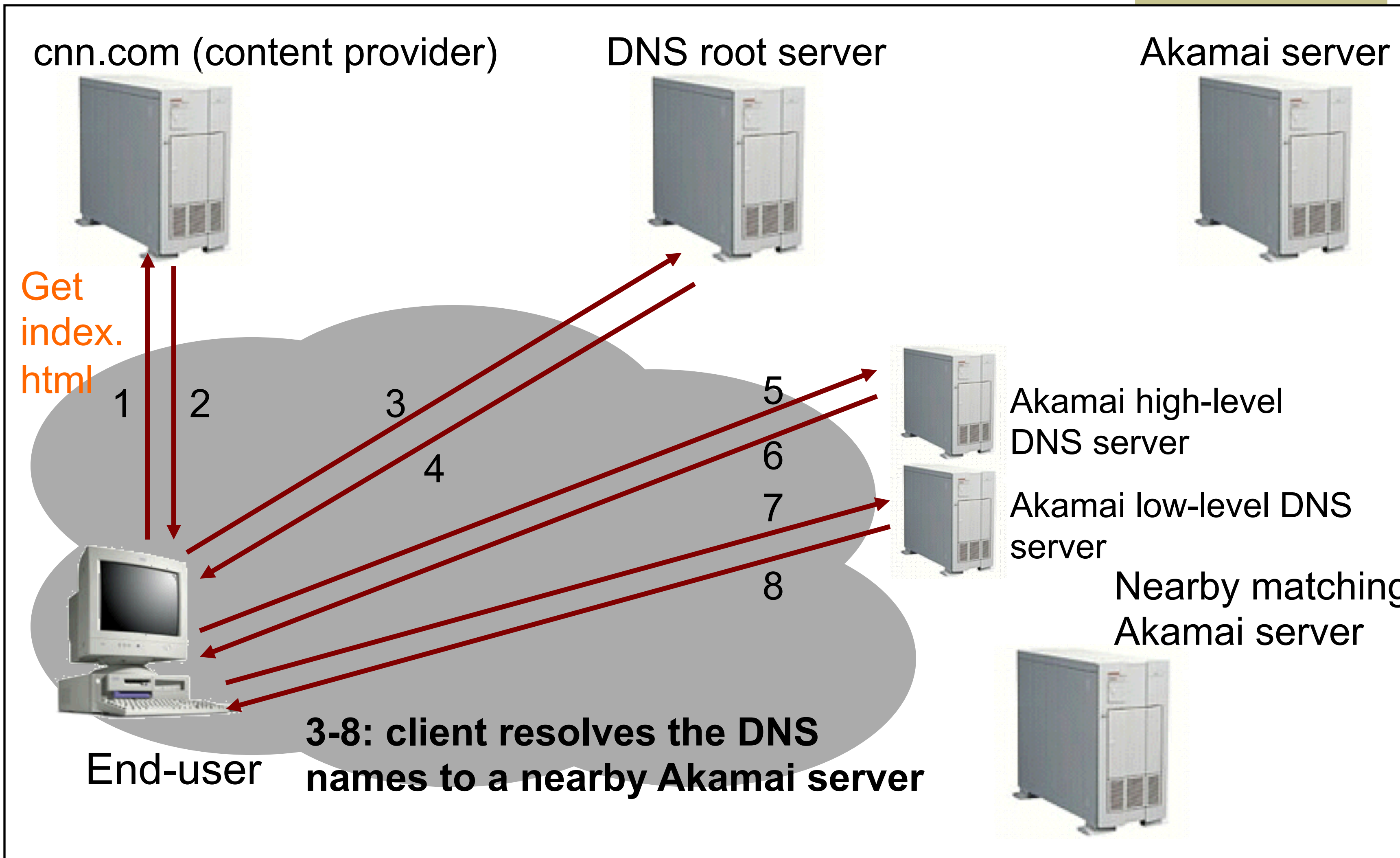
How Akamai Works – **First** time request



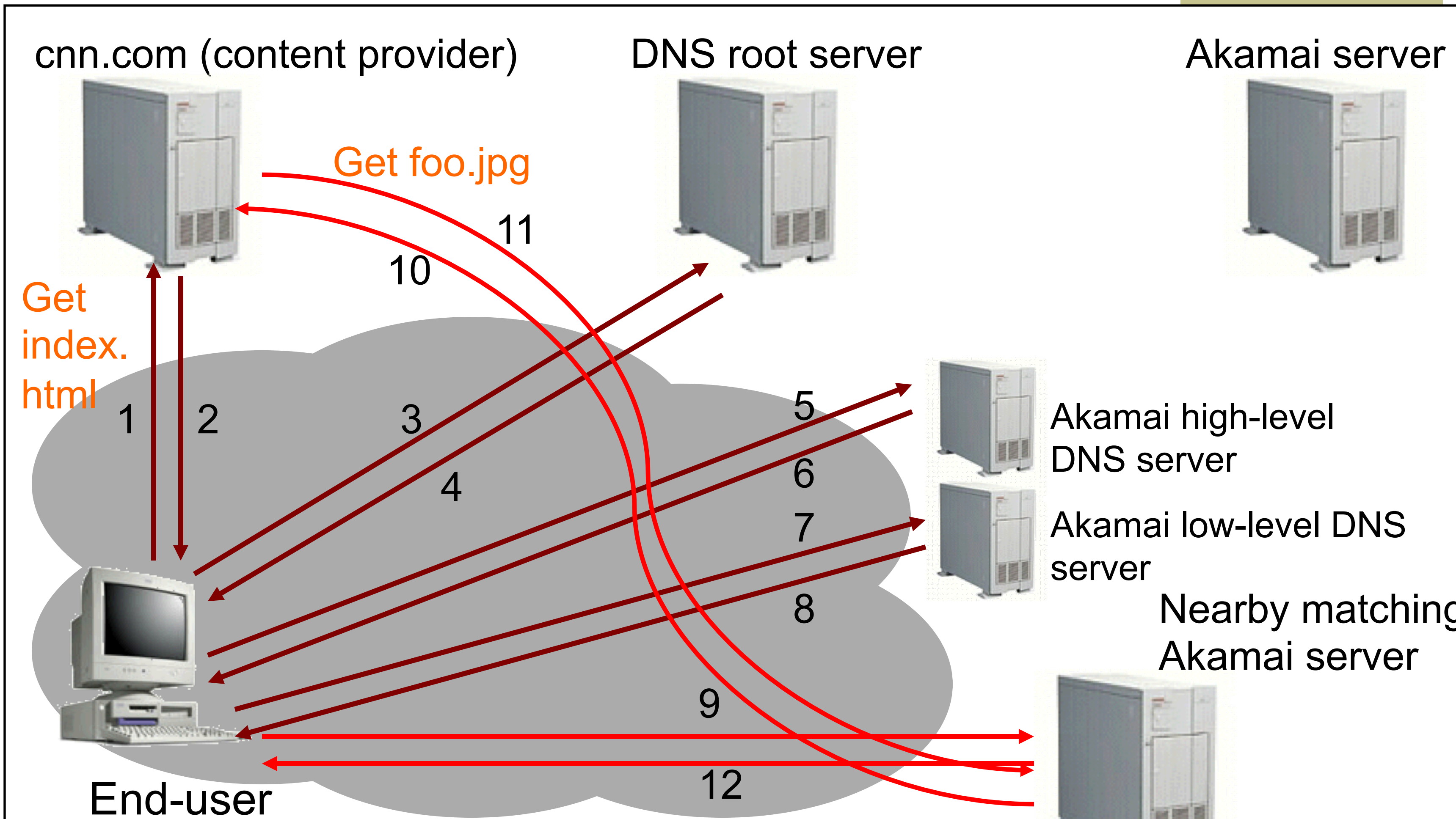
How Akamai Works – **First** time request



How Akamai Works – **First** time request

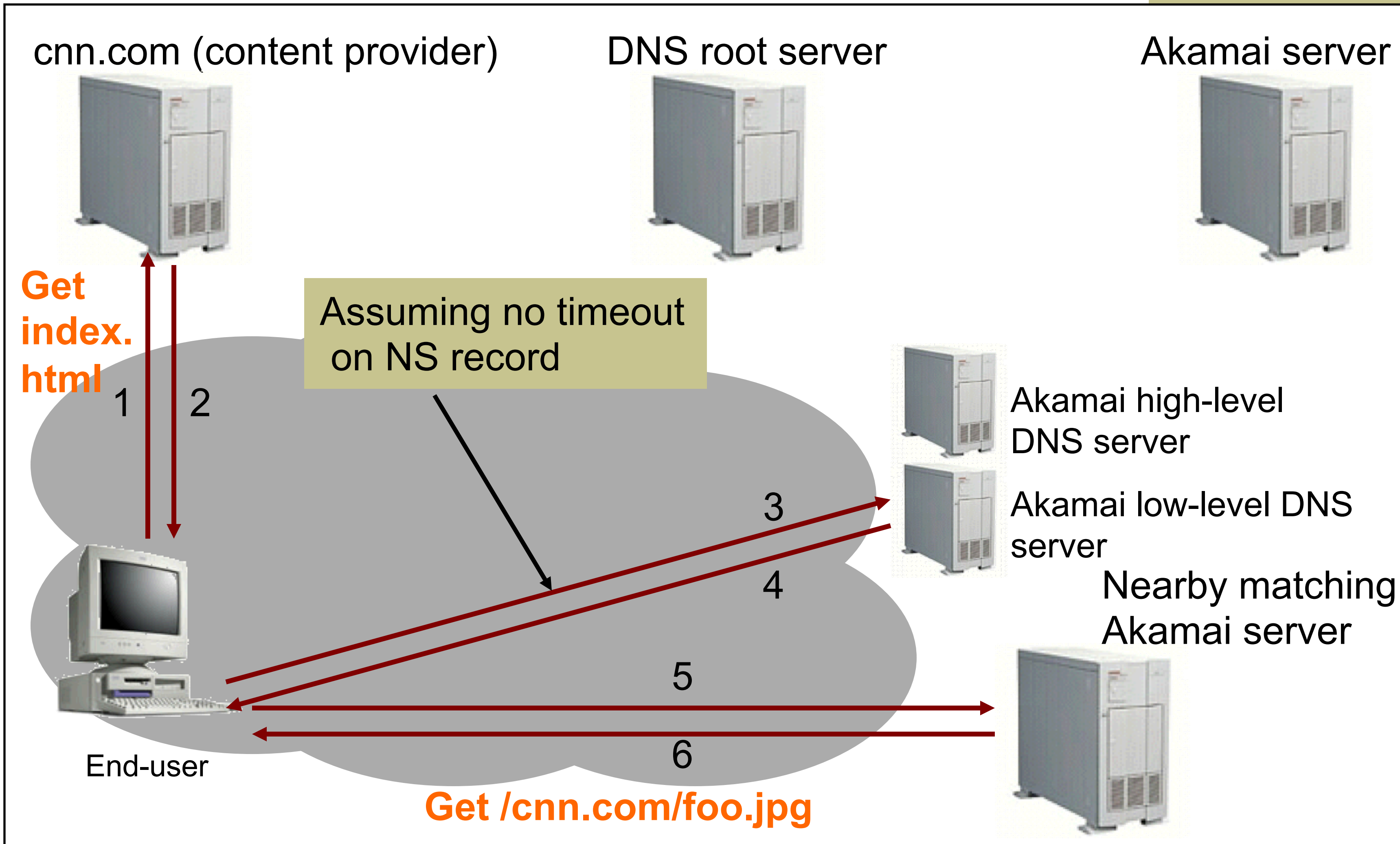


How Akamai Works – **First** time request



9-12: fetch content via proxy

Akamai – **Subsequent** Requests



Outline

- Problem with HTTP 1.1
- SPDY and HTTP2.0
- DNS Design (covered in 317)
- Content Distribution Networks
- Consistent hashing

Simple Hashing

- Given document XYZ, we need to choose a server to use
- Suppose we use modulo
- Number servers from 1…n
  - Place document XYZ on server (XYZ mod n)
    - (i.e., Placement only based on server identities)
  - What happens when a servers fails? n → n-1
    - Same if different people have different measures of n
  - Why might this be bad?

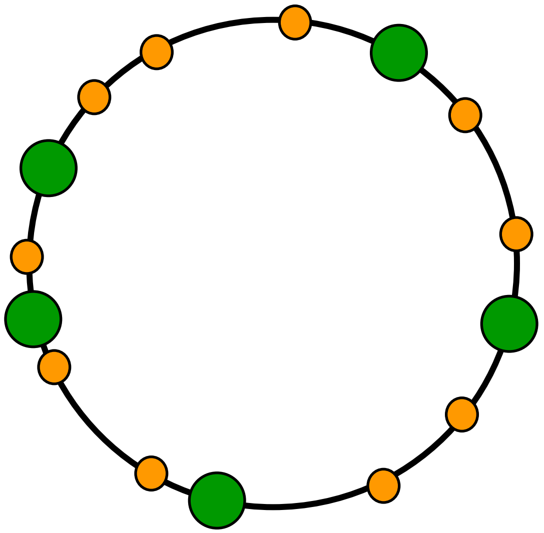
Consistent Hash

- "view" = subset of all hash buckets that are visible (a bucket is e.g., a server)
- Desired features
  - **Smoothness** – little impact on hash bucket contents when buckets are added/removed
  - **Spread** – small set of hash buckets that may hold an object regardless of views
  - **Load balance** – across all views, # of objects assigned to hash bucket is small

Consistent Hashing

- **Main idea:**
  - map both keys and nodes to the same (metric) identifier space
  - find a "rule" how to assign keys to nodes

Ring is one option.



Consistent Hashing

- The consistent hash function assigns each node and key an *m*-bit identifier using SHA-1 as a base hash function

- **Node identifier:** SHA-1 hash of IP address

- **Key identifier:** SHA-1 hash of key

Identifiers

- *m* bit identifier space for both keys and nodes

- **Key identifier:** SHA-1(key)

  Key="LetItBe" —SHA-1→ ID=60

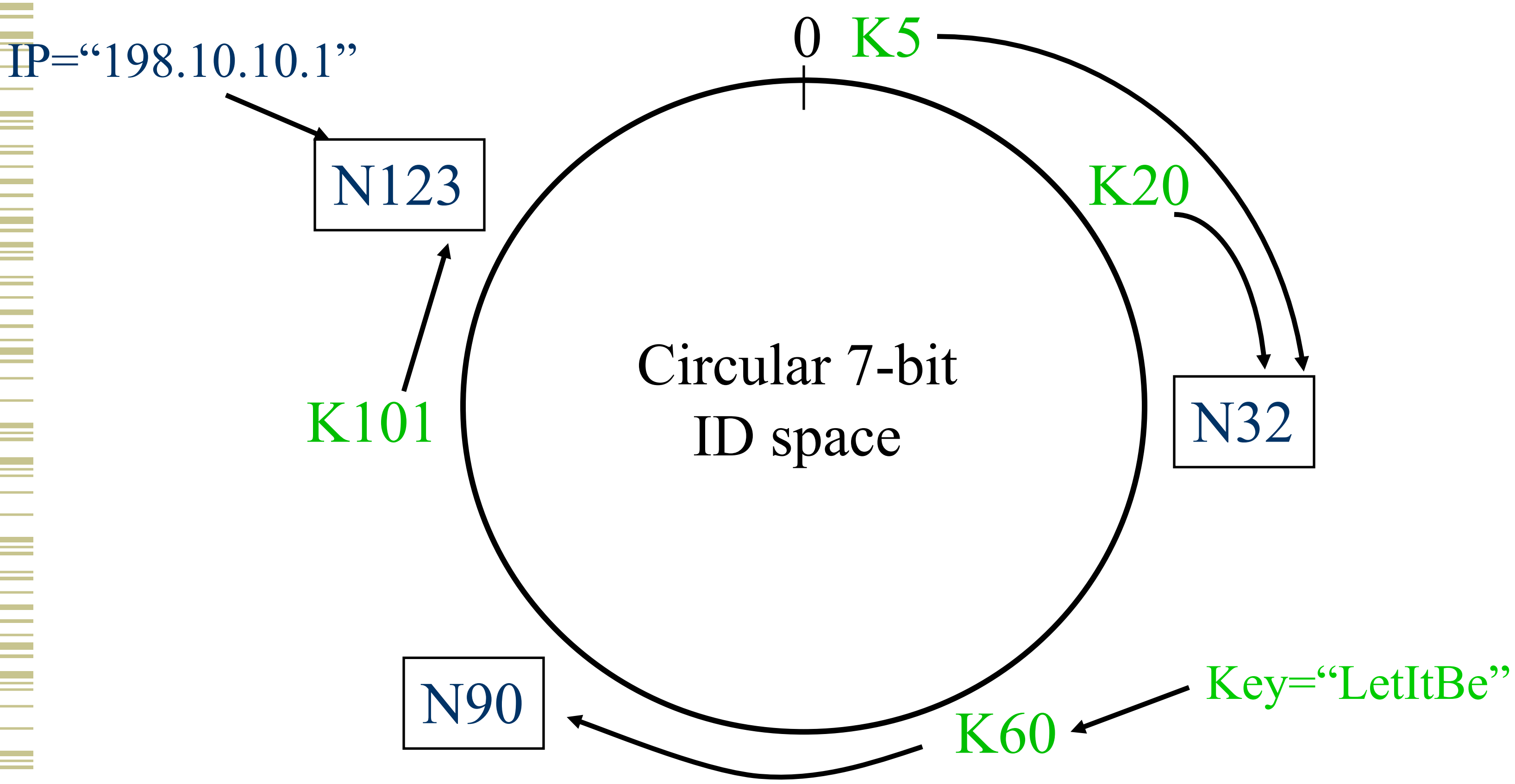
  - **Node identifier:** SHA-1(IP address)

    IP="198.10.10.1" —SHA-1→ ID=123

- How to map key IDs to node IDs?

Consistent Hashing Example

Rule: A key is stored at its successor: node with next higher or equal ID



Consistent Hashing Properties

- Smoothness → addition of node does not cause movement of objects between existing nodes
- Spread → small set of nodes that lie near object (with successor rule: object at exactly 1 node)
- Load balance → all nodes receive roughly the same number of keys. For *N* nodes and *K* keys, with high probability
  - each node holds at most $(1+\varepsilon)K/N$ keys
  - (provided that K is large enough compared to N)

Consistent Hashing not just for CDN

- Finding a nearby server for an object in a CDN uses centralized knowledge.
- Consistent hashing can also be used in a distributed setting
- P2P systems like BitTorrent, also need a way of finding files.
  - More broadly: distributed hash tables (DHTs) for decentralized lookups
- Consistent Hashing to the rescue
  - Need a way to route in a decentralized way between nodes; but easy to come up with a distance metric!

Issues with HTTP caching

- Caching (with a CDN) is nice but…
- Over 50% of all HTTP objects are uncacheable – why?
- Challenges:
  - Dynamic data → stock prices, scores, web cams
  - “CGI” scripts → results based on passed parameters
  - SSL → encrypted data is not cacheable
  - Cookies → results may be based on passed data
  - Hit metering → owner wants to measure # of hits for revenue, etc.

Summary

- Slow web with HTTP 1.1
- SPDY and HTTP 2.0 (change the app layer protocol!)
- Content Delivery Networks move data closer to user, maintain consistency, balance load
  - Consistent hashing maps keys AND buckets into the same space
  - Consistent hashing can be fully distributed, useful in P2P systems using structured overlays

More: "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web"